# A Logical Controller Architecture for Network Security

by

Yuanjun Yao

Department of Computer Science
Duke University

Date: ____________________
Approved:

____________________
Jeff Chase, Advisor

____________________
Bruce Maggs

____________________
Kamesh Munagala

____________________
Paul Ruth

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2020

# Abstract

## A Logical Controller Architecture for Network Security

by

Yuanjun Yao

Department of Computer Science
Duke University

Date: ____________________
Approved:

____________________
Jeff Chase, Advisor

____________________
Bruce Maggs

____________________
Kamesh Munagala

____________________
Paul Ruth

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2020

Copyright © 2020 by Yuanjun Yao
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

# Abstract

Networked infrastructure-as-a-service testbeds are evolving with higher capacity and more advanced capabilities. Modern testbeds offer stitched virtual circuit capability, programmable dataplanes with software-defined networking (SDN), and in-network processing on adjacent cloud servers. With these capabilities they are able to host virtual network service providers (NSPs) that peer and exchange traffic with edge subnets and with other NSPs in the testbeds. Testbed tenants may configure and program their NSPs to provide a range of functions and capabilities. Programmable NSPs enable innovation in network services and protocols following the pluralist philosophy of network architecture.

Advancing testbeds offer an opportunity to harness their power to deploy production NSPs with topology and value-added features tailored to the needs of specific user communities. For example, one objective of this research is to define abstractions and tools to support built-to-order virtual science networks for data-intensive science collaborations that share and exchange datasets securely at high speed. A virtual science network may combine dedicated high-speed circuits on advanced research fabrics with integrated in-network processing on virtual cloud servers, and links to exchange traffic with customer campus networks and/or testbed slices. We propose *security-managed* science networks with additional security features including access control, embedded virtual security appliances, and managed connectivity according to customer policy. A security-managed NSP is in essence a virtual *software-defined*

*exchange* (SDX) that applies customer-specified policy to mediate connectivity. This dissertation proposes control abstractions for dynamic NSPs, with a focus on managing security in the control plane based on programmable security policy. It defines an architecture for automated *NSP controllers* that orchestrate and program an NSP's SDN dataplane and manage its interactions with customers and peer NSPs. A key element of the approach is to use declarative trust logic to program the control plane: all control-plane interactions—including route advertisements, address assignments, policy controls, and governance authority—are represented as signed statements in a logic (trust datalog). NSP controllers use a logical inference engine to authorize all interactions and check for policy compliance.

To evaluate these ideas, we develop the ExoPlex controller framework for secure policy-based networking over programmable network infrastructures. An ExoPlex NSP combines a logical NSP controller with an off-the-shelf SDN controller and an existing trust logic platform (SAFE), both of which were enhanced for this project. Experiments with the software on testbeds—ExoGENI, ESnet, and Chameleon—demonstrate the power and potential of the approach. The dissertation presents the research in four parts.

The first part introduces the foundational capabilities of research testbeds that enables the approach, and presents the design of the ExoPlex controller framework to leverage those capabilities for hosted NSPs. We demonstrate a proof-of-concept deployment of an NSP with network function virtualization, an elastic dataplane, and managed traffic security on the ExoGENI testbed.

The second part introduces logical trust to structure control-plane interactions and program security policy. We show how to use declarative trust logic to address the challenges for managing identity, resource access, peering, connectivity and secure routing. We present off-the-shelf SAFE logic templates and rules to demonstrate a virtual SDX that authorizes network stitching and connectivity with logical trust.

The third part applies the controller architecture to secure policy-based *inter-domain* routing among *transit NSPs* based on a logical trust plane. Signed logic exchanges propagate advertised routes and policies through the network. We show that trust logic rules capture and represent current and evolving Internet security protocols, affording protection equivalent to BGPsec for secure routing and RPKI for origin authentication. The logic also supports programmable policy for managed connectivity with end-to-end trust, allowing customers to permission the NSPs so that customer traffic does not pass through untrusted NSPs (path control).

The last part introduces SCIF, which extends logical peering and routing to incorporate customizable policies to defend against packet spoofing and route leaks. It uses trust logic to define more expressive route advertisements and compliance checks to filter advertisements that propagate outside of their intended scope. For SCIF, we extended the ExoPlex SDN dataplanes to configure ingress packet filters automatically from accepted routes (unicast Reverse Path Forwarding or uRPF). We present logic templates that capture the defenses of valley-free routing and the Internet MANRS approach based on a central database of route ingress/egress policies (RADb/RPSL). We show how to extend their expressive power for stronger routing security, and complement it with path control policies that constrain the set of trusted NSPs for built-to-order internetworks.

# Contents

**Abstract** **iv**

**List of Tables** **xi**

**List of Figures** **xii**

**Acknowledgements** **xiv**

**1 Introduction** **1**

1.1 Thesis: Logical Trust for Programmable NSPs . . . . . . . . . . . . . 4

1.2 Security-Managed Virtual SDX . . . . . . . . . . . . . . . . . . . . . 5

1.3 Logical Peering for Interdomain Networking . . . . . . . . . . . . . . 7

1.4 Internet Filtering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.5 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

1.6 Structure of the Dissertation . . . . . . . . . . . . . . . . . . . . . . . 10

**2 Related Work** **11**

**3 ExoPlex Controller** **16**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.2 Slice control and Cross-Slice Stitching . . . . . . . . . . . . . . . . . . 17

3.2.1 Background: Networking on ExoGENI . . . . . . . . . . . . . 19

3.2.2 Slice Stitching in ExoGENI . . . . . . . . . . . . . . . . . . . 21

3.2.3 Instantiating and Connecting Slices Programmatically . . . . . 24

3.2.4 Demonstration Experiments . . . . . . . . . . . . . . . . . . . 27

3.2.5 Network Stitching across Research Testbeds . . . . . . . . . . 31
3.3 ExoPlex . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.3.1 Software Architecture . . . . . . . . . . . . . . . . . . . . . . . 33
3.4 Virtual SDX: a Prototype NSP . . . . . . . . . . . . . . . . . . . . 36
3.4.1 Elastic Traffic Monitoring Service . . . . . . . . . . . . . . . . 38
3.4.2 Elastic Network Transit Service . . . . . . . . . . . . . . . . . . 44
3.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
**4 Logical Trust for Multidomain Networking 47**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 Foundations: SAFE Logical Trust . . . . . . . . . . . . . . . . . . . 48
4.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.2.2 Off-the-Shelf Logic Rules . . . . . . . . . . . . . . . . . . . . . 52
4.2.3 SAFE Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.3 Logical Trust in ExoPlex . . . . . . . . . . . . . . . . . . . . . . . 60
4.4 Logical Trust for a Virtual SDX . . . . . . . . . . . . . . . . . . . . 62
4.5 Evaluating Logical Trust for Virtual SDX . . . . . . . . . . . . . . . 65
4.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
**5 Logical Peering for Interdomain Networking on Testbeds 69**
5.1 Logical Peering in ExoPlex . . . . . . . . . . . . . . . . . . . . . . 70
5.2 Design Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2.1 Secure Routing . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2.2 Path Control . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2.3 Policy Composition and Priority . . . . . . . . . . . . . . . . . . 74
5.2.4 Processing Advertised Routes and Policies . . . . . . . . . . . 76
5.2.5 Source-specific Routing with SDN-enabled Dataplane . . . . . 82

5.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.3.1 Control Plane . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.3.2 Example Scenario: FabNet . . . . . . . . . . . . . . . . . . . 83
5.3.3 Secure Routing with Logical Trust . . . . . . . . . . . . . . . 84
5.3.4 Authorization for the NSP API . . . . . . . . . . . . . . . . . 85
5.3.5 Route Validation . . . . . . . . . . . . . . . . . . . . . . . . 85
5.3.6 Discussion: Governance . . . . . . . . . . . . . . . . . . . . . 86
5.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.4.1 Experiment 1: Inbound Path Control . . . . . . . . . . . . . . 89
5.4.2 Experiment 2: Outbound Path Control . . . . . . . . . . . . . 90
5.4.3 SAFE Routing Authorization Performance . . . . . . . . . . . 91
5.4.4 Prefix Pair Matching with AQT . . . . . . . . . . . . . . . . . 94
5.4.5 Dataplane Overhead for Source-Specific Routing . . . . . . . . 94

**6 Secure Customizable Internet Filtering 99**

6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.2.1 Internet Defenses for Spoofed Traffic . . . . . . . . . . . . . . 103
6.2.2 Routing Policies of NSPs . . . . . . . . . . . . . . . . . . . . 106
6.2.3 Limiting the Adversary: Trust Model . . . . . . . . . . . . . . 106
6.2.4 Logical Routing Security . . . . . . . . . . . . . . . . . . . . . 107
6.3 SAFE Customizable Internet Filtering . . . . . . . . . . . . . . . . 108
6.3.1 Capturing RPSL with Logic . . . . . . . . . . . . . . . . . . . 109
6.3.2 Customer Routing Policy . . . . . . . . . . . . . . . . . . . . . 111
6.3.3 Routing Policy Delegation . . . . . . . . . . . . . . . . . . . . 112
6.3.4 Routing Policy with Groups . . . . . . . . . . . . . . . . . . . 113

6.3.5 Expressing MANRS with SCIF . . . . . . . . . . . . . . . . . . 114
6.3.6 Improving MANRS . . . . . . . . . . . . . . . . . . . . . . . . 115
6.3.7 Route Propagation and Ingress Filtering . . . . . . . . . . . . 116
6.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
**7 Discussion** **119**
7.1 Datalog for Network Policies . . . . . . . . . . . . . . . . . . . . . . 119
7.2 Compensations and Motivations for NSPs . . . . . . . . . . . . . . . . 120
7.3 Intradomain Traffic Engineering for NSPs . . . . . . . . . . . . . . . 121
**8 Conclusion** **122**
**Biography** **132**

# List of Tables


3.1 Control plane APIs of an NSP controller. . . . . . . . . . . . . . . . . . . . . 34

3.2 Required, provisioned and measured bandwidth between tenant subnets. . . 45

4.1 Exemplary policies for network authorization in ExoPlex and their complexity and authorization throughput. . . . . . . . . . . . . . . . . . . . . . . . . 64

5.1 NSP controller data structures used by Algorithm 1 and Algorithm 2. . . . 76

5.2 The states of NSP 5 before and after processing the outbound policy from 6 in Figure 5.3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

5.3 The inbound and outbound path control policies of subnets in Figure 5.1. . 90

5.4 The path for traffic between subnet pairs at each step. . . . . . . . . . . . . 91

# List of Figures


3.1 An L3 network transit service in a slice. . . . . . . . . . . . . . . . . . . . . . 22

3.2 Example message flow for slice-to-slice stitching. . . . . . . . . . . . . . . . . 22

3.3 A dumbbell topology for evaluating the network transit and exchange service for GENI slices. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.4 A shared link in a carrier slice is shared fairly among flows from multiple customer sender/receiver pairs. . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.5 A single bottleneck link shared by multiple customer sender/receiver pairs. 31

3.6 A demonstration experiment: connecting client networks at Chameleon and ExoGENI with a VFC network in ESnet. . . . . . . . . . . . . . . . . . . . 32

3.7 An exemplary ExoPlex Network Service Provider (NSP). . . . . . . . . . . 35

3.8 NFV controller in ExoPlex that scales Bro pool via slice controller and set traffic mirroring rules via SDN controller. . . . . . . . . . . . . . . . . . . . 39

3.9 An experimental SDX network that exchanges traffic between customer nodes and mirrors traffic to Bro nodes for intrusion detection. . . . . . . . . . . . 41

3.10 Bro performance in ExoPlex experiments. . . . . . . . . . . . . . . . . . . . 42

3.11 An NSP provides network transit service with bandwidth requirement to tenant networks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.1 Set linking examples for STRONG nested groups and SAFE routing. . . . . 57

4.2 Comparison of inference cost for group delegation between noisy and compact contexts, with primary and secondary indexing schemes. . . . . . . . . . . . 57

4.3 Raw inference time for access checks against an ACL of groups, as a function of ACL length and group delegation depth. . . . . . . . . . . . . . . . . . . 59

4.4 Latency of routing verification under three query workload models. . . . . 60

4.5 Linking patterns for SAFE sets/certificates in the GENI-derived trust model with attribute service. . . . 63

4.6 Ping results between subnets. . . . 67

5.1 An exemplary inter-domain network used in our experiments. . . . 71

5.2 Cases for two overlapping prefix pairs. . . . 75

5.3 An example scenario in which policy overlap forces selection of an alternate route for the intersection. . . . 80

5.4 Flow tables of NSP switches in experiment 1. . . . 89

5.5 Timeline of the experiment for outbound path control. . . . 91

5.6 Flow tables of NSP switches in experiment 2 after step 6. . . . 92

5.7 SAFE logic engine throughput to validate routes for different logical policies. 93

5.8 Processing time for each prefix pair insertion, deletion and query operation with AQT. . . . 95

5.9 The CPU utilization and the number of sampled megaflows with different numbers of flows and different numbers and types of OpenFlow entries. . . . 97

6.1 An example route leak: a multi-homed network $B$ leaks the route it learned from its provider $A$ to another provider $C$. . . . 105

6.2 Accepted routes by BGPsec, RPSL, MANRS and more restrictive SCIF policies. 110

6.3 NSPs must prefer routes learned from customers over routes learned from peers/providers for ingress filtering. . . . 114

6.4 An exemplary customized routing policy with privilege delegation. . . . 117

# Acknowledgements

This dissertation would have been impossible without my advisor Jeff Chase. It has been a great experience to pursue my PhD study with him. Jeff has always been supporting, inspiring and insightful with his knowledge and experiences. Discussions with him led to sparkles of innovations that evolved into solutions for the research problems. Jeff gave me freedom to do independent research while providing valuable suggestions, sharp insights and hands-on assistance. Thanks to Jeff for his mentorship and advice on my research and my career.

Thanks to Qiang for his guidance, help and collaborations on SAFE, logical policies and ExoPlex. He is also motivating with his rigorous attitude towards research and hard working. Thanks to Rubens Farias for his contributions to the integration of Bro with ExoPlex.

I would like to thank our collaborators, Paul Ruth, Mert Cevik, Cong Wang, Komal Thareja, Ilya Baldin, Yufeng Xin and Anirban Mandal from RENCI, and Nick Buraglio from ESnet for their support with testbeds including ExoGENI, Chameleon and ESnet. The stitching capabilities they brought to the testbeds are foundational for experimental interdomain networking in testbeds. I would like to thank Paul for his hard work and support with Ahab, ExoGENI, and Chameleon network and his precious suggestions with ExoPlex and SDN controllers for Corsa switch VFCs. Thanks to Mert for his expertise and support with the testbeds, including setting up VFC networks in ESnet and stitching them to Chameleon networks and ExoGENI networks. Thanks to Viktor Orlikowski for his work on stitching Duke campus SDN network to ExoGENI that extends ExoPlex for campus

networks. Thanks to Cong and Komal for using ExoPlex and providing feedback.

Thanks to Professor Bruce Maggs, Professor Kamesh Munagala and Doctor Paul Ruth for serving as my committee members and for their feedback on my research projects and milestones.

Thanks to my mentors and managers Wei Guo, Yang Ping, Mihir Patil, Madhu Vohra and Mary Firenze for their help and support during my internship at VMware. I was also fortunate to work and talk with Eric Kao, Harold Lim, Leonid Ryzhyk and Mihai Budiu there.

Thanks to my lab mates and friends Bing Xie, Yuhao Wen, Zhengjie Miao, Shengbao Zheng, Junyang Gao, Yuan Deng and Xi He for bringing joy to my life.

Last but not least, I would like to thank my wife Yameng Liu for her motivations, support and companion during the journey. Thanks to my family and Yameng's family for their support.

This material is based upon work supported by the National Science Foundation under Grants No. (ACI-1642140, ACI-1642142, CNS-1330659, CNS-1243315) and through the Global Environment for Network Innovations (GENI) program. Any opinions, findings, and conclusions or recommendations do not necessarily reflect the views of NSF.

# 1

# Introduction

The Internet was designed to be open and transparent when it started from a small group of users with common motivations and shared trust. As the Internet grows, it opens to more and more players with diverse interest, leading to the "tussle" among different players competing for their own self-interest or adverse interest [28]. For example, attackers may launch network attacks for their own benefits while harming others. A network service provider (NSP)[1] may pass traffic off to another NSP as soon as possible ("hot-potato" routing) rather than choosing the best path to the destination. People have proposed various approaches to address the security threats and improve services in the Internet, but NSPs may be reluctant to deploy new protocols or new services when they benefit little from the incremental deployment.

To create opportunities for new approaches to existing problems in the Internet, and to innovate new network protocols, network services and applications, some researchers proposed ideas for "pluralist" network architecture based on deep virtualization. Most notably, the authors of Plutarch [29] and Cabo [34, 16], propose to build concurrent and heterogeneous NSPs as software layers over programmable infrastructures: pipes, programmable

[1] In this dissertation we refer to the participants in inter-domain networking generically as network service providers, which are also called autonomous systems in the Internet.

switching points, and in-network computation.

They offer a compelling vision of NSPs that manage end-to-end interoperable connectivity, riding over an architecture-neutral underlay of infrastructure providers. It allows NSPs to provide various network services with virtual networks allocated from infrastructure providers, giving more choices for customers, which is in line with the "design for choice" principle to deal with the tussles in the Internet. Researchers haven't reached an agreement on what should be the right network architecture, or even if it is necessary to have a network architecture [75]. The idea of "pluralist" network architecture is inspiring and interesting.

Various network testbeds and advanced research fabrics have been created to facilitate disruptive innovation in the Internet architecture and facilitate the deployment of new services and protocols. Networked Infrastructure-as-a-Service (NIaaS) testbeds are evolving with higher capacity and more advanced capabilities. Modern network testbeds offer stitched virtual circuits, programmable dataplanes, and in-network processing on adjacent cloud servers, making it possible to run NSPs with dedicated virtual networks (slices) that peer and exchange traffic with edge subnets or other NSPs in the testbeds.

The network testbeds can serve as a blueprint for future commercial deployments of NIaaS platforms that host NSPs. With these advanced testbed capabilities, we can build and deploy NSPs following the pluralist philosophy of Plutarch and Cabo. These NSPs can serve edge subnets (e.g., campus networks) or other networks hosted on the testbeds.

One goal of this research is to define the abstraction and tools to support built-to-order virtual science network on network testbeds for data-intensive science collaborations. The virtual science network connects multiple science networks on campus and/or network testbeds with bandwidth-provisioned links, integrated in-network security monitoring and managed connectivity. A security-managed virtual science network is in essence a virtual software defined exchange (SDX) that applies customer-specified policy to mediate connectivity.

More generally, we can deploy various NSPs on network testbeds to serve experimental

or production traffic. And NSPs can peer with other NSPs to provide network services jointly, ending up with internetworks on testbeds that are similar to the Internet. The NSPs on network testbeds may also peer with NSPs in the Internet and weave into the fabric of the Internet over time through cycles of innovation and adoption.

A second goal of this research is to leverage testbed-hosted NSPs as a model for interdomain networking in the Internet. In the Internet today, a network service provider runs its service network with its own network infrastructure as an Autonomous System (AS) and peers with other ASes for global connectivity. Experimenting with NSPs on network testbeds frees us from the specific constraints of the routing protocols (BGP) that have evolved over time in the Internet, including security protocols and practices that are as yet incompletely deployed. It opens the possibility to experiment with new and more flexible abstractions for programming trust in interdomain networks. While we conduct this research apart from the "real" Internet, it enables us to experiment with approaches and techniques that can be applied for stronger and more flexible security in the future Internet.

With the enabling testbed capabilities at hand, this research focuses on the control abstractions to program NSPs so that their footprint and interactions may change over time according to the needs of their supported communities and traffic. To protect them, the programming model must incorporate all of the security features of the modern Internet, and also permit flexible control of security policy, including enhanced security features such as in-network inspection, access control, connectivity management, path control, and anti-spoofing protections. And programming trust is the core of the problem.

In this dissertation, we propose the ExoPlex logical controller framework for running virtual NSPs on network testbeds. It provides network transit services with such add-on values as network function virtualization (NFV), quality of service (QoS) and managed security. The ExoPlex controller framework has different service modules layered above platform-specific slice controllers, SDN controllers to manage various network services and a logical trust platform to manage trust and security for multidomain networking. The logical trust platform accommodates flexible, innovative and customized network policies

that are deployable in testbed-hosted NSPs with ExoPlex. ExoPlex allows us to evaluate existing approaches for Internet security with NSPs in the testbeds and to explore new approaches for stronger network security that the current Internet may not afford. For example, we propose secure customized interdomain routing with secure routing policies and customized path control policies that allows the customers to control what NSPs can carry their traffic. We extend that idea to Secure Customizable Internet Filtering (SCIF) for secure routing and anti-spoofing with ingress filtering.

## 1.1 Thesis: Logical Trust for Programmable NSPs

With these platforms in place, we can run *multi-cloud* applications that leverage benefits of scale, diversity, geographic dispersion, and heterogeneity, and embody the vision of pluralist network by running NSPs on the NIaaS testbeds. What we need is an abstraction for higher-level NSP controller that builds elastic, programmable and flexible NSPs.

We propose ExoPlex controller framework to run virtual NSPs in various testbeds with SDN-enabled IP dataplane. ExoPlex utilizes testbed-specific slice controllers to adapt the topology of the service slice according to changing demands. ExoPlex runs on top of SDN controllers to configure the dataplane to forward packets or mirror the traffic for monitoring. By separating the dataplane and control plane with SDN, ExoPlex is able to support flexible network policies in the control plane to address various security threats.

While we demonstrate the potential for elastic NSPs with add-on values like QoS and NFVs, we focus on managing the security of multidomain networking. We use a logical trust language (Datalog) to represent all security metadata, including facts, policies, endorsements and delegations. We use SAFE [22] logical trust framework for management of the security data and logical inference. SAFE [22] defines a certificate format for signed logic payloads, and a validation engine for policy checks incorporating an off-the-shelf Datalog engine (Styla [82]). The logic vocabulary is extensible and enables a wide range of policies and trust structures without changing the certificate format or platform implementation. Our approach is inspired in part by earlier work on networking using Datalog, e.g., [59, 5].

The logical trust approach is a rapid prototyping vehicle for experimental approaches to secure multidomain networking. Although the power and flexibility of trust logic imposes substantial costs as prototyped, they are off of the dataplane and accrue only on changes to the network (e.g., peer link stitching, new prefix announcements) or its security policies. Importantly, the logic approach permits but does not require participants to write logic code: they can delegate their policies to others, take prepackaged policy off the shelf (e.g., from federation authorities), or use packaged logic for common structures and access control abstractions. ExoPlex manages security data including security policies of related players with SAFE. It authenticates and enforces the security policies to make sure the network configurations are compliant to the security policies. The customization and flexibility of the security policies lead to flexible and programmable multidomain networking.

## 1.2 Security-Managed Virtual SDX

At the start of this project, we developed a security-enhanced NSP with logical trust as an illustrative example of a novel service that can be useful in a practical deployment. We developed support for built-to-order *virtual science networks* to meet the need for secure high-volume data sharing in data-intensive science communities. A virtual science network interconnects campus subnets over high-bandwidth research fabrics. It manages access and connectivity according to policy specified by the participants—a function known as Software-Defined Exchange (SDX). Our virtual SDX has the following functions and requirements:

- **Elastic dataplane.** The virtual science network dynamically adapts the slice topology to meet the changing customer demands.
- **Quality of Service.** Customer networks can reserve bandwidth for their connections (QoS).
- **NFV for flow monitoring.** The virtual science network provides fast and elastic out-of-band traffic monitoring with security appliances deployed in the network with Network Function Virtualization (NFV).

- **Access control.** Customers need to stitch their networks to the virtual SDX with layer 2 circuits in order to exchange traffic with other customer networks via the virtual SDX. The virtual SDX must authenticate the network stitching request from customer networks. Only authorized campus networks or testbed slices can stitch to the virtual SDX.
- **Prefix ownership.** For different customer networks to communicate, it requires certain governance to assign and delegate IP prefixes from a common IP space to them.
- **Managed connectivity.** Connectivity between customer networks is off-by-default and is enabled only by mutual consent (by customer policies),

We deployed the virtual science network on ExoPlex. The controller uses straw-man strategies to provision elastic network resources (e.g., links, in-network computing nodes) and to configure the elastic dataplane with SDN, QoS and NFV.

Our approach is related to other recent systems for policy-based network management. Recent network management languages (such as FSL [25], PGA [69], PANE [35], or Merlin [78]) provide declarative policy tools to manage connectivity among network endpoints. All of these languages are based on an "off by default" model for interconnection based on user-defined attributes (labels) and predicates for network endpoints. These systems are designed for use within an enterprise network with a global view of policies by all participants, with policies specified by the network owner. We apply similar ideas in a decentralized way, in which all participants may specify policies to control their own traffic.

We demonstrate how virtual SDX authorizes slice stitching requests controls connectivity between customer subnets with exemplary stitching policies and connectivity policies. To authorize a slice stitching request, the stitching policy verifies the user's control privileges over the slice and authenticates users with access control lists. The connectivity between customer subnets is off-by-default. The virtual SDX checks the customer connectivity policies and allows connectivity between customer subnets only with mutual consent and authenticated prefix ownership.

The policies of different participants can be flexible. Some policies may approve requests from others with access control lists (ACLs) that enumerate the acceptable ones. Some policies may allow requests from others who are endorsed by certain governance. And the choice of governance structure can be flexible as well, though different parties must agree on the same governance for interoperability.

## 1.3 Logical Peering for Interdomain Networking

The NSPs can also peer and exchange traffic with other NSPs, leading to interdomain networking similar to the Internet. The logical trust approach with its flexibility can be a powerful tool to secure interdomain networking as well. To understand what is necessary for interdomain network security, we look into the Internet, which is the largest interdomain networking with a long history, and learn from the security threats in the Internet and the existing approaches to addressing those threats.

In the Internet, different ASes peer and exchange traffic with one another for global connectivity. ASes transitively route a packet from the source to the destination using the Internet Protocol (IP). The Internet Protocol has developed as the "thin waist" at network level that glues various protocols at data link layer and application layer together with TCP and UDP. For different parties to communicate over IP, we need some governance structure to assign IP prefixes to NSPs and customers over hierarchical resource delegations (e.g. RPKI [21]). To discover paths to the destination IP prefixes, ASes exchange routing information with their neighbors via BGP. And to ensure the validity of routing information, people proposed to authorize a route by origin authentication and path validation that transitively validates the route advertisement for each hop that is signed under the key pair of the advertiser (e.g. as in BGPsec [80], SBGP [54]).

One of our goals is to explore and evaluate the evolving Internet security features and their limitations with internetworks on network testbeds, as we face similar security threats in interdomain networking on testbeds as in the Internet today, such as unauthorized traffic, route hijacking, IP address spoofing. As an approach to model and evaluate existing

approaches for secure Internetworking, we implement equivalent mechanisms in logic and deploy and evaluate them in testbed-hosted NSPs. The logical expression also serves as a basis to extend the security mechanisms and protocols with additional policies and protections for customized and stronger routing security.

There is need for a new architecture to control the threats and secure the interdomain routing. We propose logical peering that manages policy-based interdomain NSP networking in the ExoPlex toolkit. We provide standard logic rules for origin authentication and transitive route validation, modeling RPKI and BGPsec. As an example, we allow customer networks to specify connectivity policies that constrain what other networks can talk to them. And we allow customer networks to define the path control policies to limit which NSPs are able to carry their traffic. With path control, the transit path for each flow is compliant end-to-end with rules specified by the endpoints: the endpoints trust each NSP along the path to be faithful to the policies and to forward and accept traffic only along the trusted path, providing deep defenses against packet dropping, tampering, eavesdropping and spoofing.

## 1.4 Internet Filtering

Internet routing security has historically been weak. Routing attacks and IP address spoofing have been non-negligible threats to the Internet. Malicious NSPs can hijack or leak routes to attract traffic to their networks to drop, modify or spy on packets. Users can also send IP packets with spoofed source IP address, which is called IP address spoofing. IP address spoofing can be a weapon to launch other network attacks like DDoS attacks and disguise the attacker's real identity.

Various approaches to extend the routing security infrastructure have evolved to address these threats. The basic approach combines two forms of filtering. First, *route filtering* detects and rejects illegal routes as they propagate. Second, *ingress filtering* blocks arriving packets that might have spoofed source IP addresses. These two forms of filtering are closely related: it is common to filter incoming packets with *unicast Reverse Path Forwarding*

*(uRPF)*, which rejects packets that arrive from interfaces through which there is no accepted route back to the source.

It follows that the key challenge to defending against spoofing and hijacking is to determine whether any given imported route is "legitimate" before accepting it. In the Internet, there is RPKI service for route origin authentication and and BGPsec for route validation. Though RPKI and BGPsec are not yet fully deployed, their behavior and limitations are well-understood. These defenses are not sufficient (e.g., see [40]). By malice or misconfiguration, networks may export routes that are valid (i.e., the exporter has a certified path to the source) but are outside of their intended scope [81].

A more complete defense requires the NSPs to share policies for route export/import. Policy sharing among NSPs permits them to identify routes that are *faithful* to all relevant policies, and filter routes that are unfaithful, i.e., are not compliant with policies of predecessor NSPs in a path. The leading example of this approach is MANRS [3], which promotes sharing of route import/export policies through the Internet Routing Registry (RADb) [72]. But the registry-based approach also has its limitations including incomplete and unauthenticated data and lack of authorization (§6.2).

In this dissertation, we understand route legitimacy from the perspective of trust. We propose Secure Customizable Internet Filtering (SCIF) that protects route propagation with customized routing policies and conducts uRPF-based ingress filtering. Customer networks and NSPs specify their routing policies in logical trust that are deployable with ExoPlex in virtual NSPs so that routes are propagated only to "legitimate" receivers within their intended scope.

## 1.5 Contributions

The first contribution of this dissertation is the ExoPlex controller framework for running NSPs with elastic dataplane, NFV, QoS and managed security on NIaaS testbeds. We also share our experiences with developing virtual science networks across platforms. Our collaborators from RENCI are using ExoPlex to manage secure network transit service with

slices on testbeds for data-driven science projects.

The second contribution is the logical trust framework we proposed to address the challenges in managing identity, resource access, naming, and network policies for multidomain networking. We also share the lessons we learned from our practices in using logical trust for secure networking.

The third contribution is to explore the strengths and limitations of evolving approaches for Internet routing security by expressing them within a logical formulation that is precise and rigorous.

The fourth contribution is to propose a logical model for customized routing policies that allows customer networks to control what NSPs can carry their traffic. We made customer path control policies deployable on ExoPlex that enforces the compliance check of routes against both the path control policies of the source and destination.

The last contribution is SCIF that extends logical peering and routing to incorporate customizable policies to defend against packet spoofing and route leaks. SCIF provides a uniform policy framework that captures existing approaches for the Internet security, while allowing for stronger routing security for internetworks with customizable policies.

## 1.6 Structure of the Dissertation

We organize the rest of this dissertaion as follows:

Chapter 2 describes the related work in pluralist network architecture, network policy management, software defined exchange, and Internet routing security.

Chapter 3 describes the design, implementation and evaluation of ExoPlex NSP controller architecture.

Chapter 4, 5, and 6 describe logical approaches for multidomain networking, logical peering in interdomain networking and secure customizable Internet filtering respectively.

Discussions and future work are in Chapter 7. A summary of this dissertation is in Chapter 8.

# 2

# Related Work

**Pluralist network architectures.** Researchers proposed the idea of deploying network services on overlay testbeds when designing Planetlab [66]. Plutarch [29] is an internetworking architecture that makes heterogeneity explicit. It allows inter-operation and establishes connectivity between heterogeneous networks with different network protocol suites. Cabo [34] adopts and develops the pluralist philosophy by decoupling service providers and infrastructure providers. Services providers can deploy customized or innovative network protocols and services with leased physical infrastructures from one or more infrastructure providers. This dissertation explores the pluralist philosophy by running NSPs on advanced network testbeds experimenting with new network services and network policies with testbed-hosted NSPs.

**Authorization for network security.** Authorization is important for secure multidomain network services. Authorization-based approaches have been introduced to networks to mitigate network attacks, with authenticated statements as a basis to validate and authenticate network requests. RPKI [21] is a specialized public key infrastructure that provides service to users to certify their Internet number resources (AS numbers and IP prefixes). IP prefixes are delegated from Internet Assigned Numbers Authority (IANA), to

Internet registries (regional Internet registries, national Internet registries or local Internet registries), and then to end users with certificates certifying the delegation. The regional Internet registries provide an RPKI validator [4] for users to make queries and verify route announcements.

SBGP [53] and BGPsec [80] leverage RPKI and attestations to defend against BGP hijacking attacks. The IP address block holder makes statements called Route Origin Authorizations (ROA) that authorize autonomous system (AS) to originate routes for a particular prefix or a set of prefixes owned by the holder. An authorized advertisement is either made by the origination AS authorized by the owner with ROA, or by an AS that has received an authorized advertisement from other ASes. In [5], the authors provide an example policy in SeNDlog logic for an authenticated path vector protocol. In this dissertation, we implement equivalent mechanisms for prefix ownership and route validation with logical trust. In addition, we explore customized routing policies for enhanced security.

GENI [17, 19, 63] is a federation of autonomous IaaS providers ("aggregates") linked by various trust relationships and agreements. GENI serves a community of registered researchers with various institutional and project affiliations. Each provider has various policies governing client access. These policies consider endorsements and delegations of trust among the participants, including a root trust anchor that certifies the aggregates and various authority services to govern membership and coordination. In this respect GENI is representative of federated cloud systems in general, although there are differences in terminology. The trust structure of GENI and other federated cloud systems is fundamental for authorization of multidomain interactions on network testbeds. And we implement the GENI trust with logic similarly in this dissertation.

**Network policy management.** Researchers have proposed various systems and policy languages to manage security and functions of networks with network policies. Ethane [25] proposes a flow-based security language (FSL) that is Datalog-based to specify rules for mapping unidirectional flows to security constraints on them in enterprise networks. Netquery [77] is a knowledge plane for federated networks that bases trustworthiness on a

trusted platform module that allows remote authentication and reasoning about network knowledge. Merlin [78] supports network policies in logical predicates and regular expressions and sub-policy delegations. It compiles the policies into a linear programming model for forwarding paths, transformation placement and bandwidth allocation problems. Anzere [74] supports policy-based replication of personal data in a personal cloud. Users can define replication policies with resource requirements; Anzere composes the policies and solves it considering the demands and available resources with constraint logic programming. PGA [69] allows different participants in a single trust domain to write network policies independently without policy delegation, then uses graph composition to compose the policies and reconcile conflicts. Composition constraints can be added to a policy about allowed policy changes when composed with any other policies. [65] proposed a policy framework, ICING, that enables secure communication along authenticated paths among multiple network domains (realms). In PANE [35], the authors proposed the delegation of read and write privileges from the SDN controller to users, by taking requests in a specialized logic language.

SDX [42] is an SDN-enabled exchange point that allows participating NSPs to control the traffic exchange with their own policies to ensure the isolation among different participants. FLANC [41] is an authorization logic proposed to represent participants' actions on flowspace in SDX. FLANC restricts allowed sets of actions by restricting allowable part of destination flowspace with policies in logic. There is another resource delegation framework for software-defined networks that uses logic to state facts, such as ownership of flowspace and delegation of flowspace [11].

Those systems try to manage network policies in different applications with specified policy languages. In this dissertation, we apply logic-based delegations and authorizations for multidomain network security, but in a unified policy language (Datalog).

**Routing path control.** There has been research in flexible Internet routing that allows different players including the sender, the receiver and the NSP to control the routing path flexibly. [28] argues that the Internet should support source routing that allows the sender

to control the path for its packets at the provider level. [93] allows the access router of an edge network to maintain multiple routes to every destination for resilience. Loose source routing [8] enables the sender to specify transmit policies on the domain-level path for its packets and the receiver to specify receive policies to accept, block or rate-limit packets according to the path. NIRA [89] allows users to choose the sequence of providers for their packets. MIRO [88] gives transit ASes control over traffic traversing their networks. It allows an upstream AS (close to sender) to negotiate with downstream ASes to discover and use alternative routes and the downstream AS can also negotiate with upstream ASes to control its incoming traffic. Wiser [60] allows ISPs to find efficient end-to-end paths and improve traffic engineering jointly. Bolero [84] separates routing policies from the path vector system and supports policy innovations by leveraging SDN. It specifies policy in logic and allows downstream path control with logic integrity constraints.

**Network accountability.** Network accountability is the ability to identify the sender of a packet by its source address (e.g. source IP address for an IP packet). However, a misbehaving sender may spoof the source address to disguise its identity and launching other attacks with source address spoofing. [36] proposes ingress filtering on periphery routers to prevent spoofing, which is light-weight but marginally effective, as it is challenging to build efficient ingress filters and spoofed traffic can still be injected from compromised networks and networks without appropriate ingress filtering. MANRS [3] recommends construcing packet filters with unicast Reverse Path Forwarding (uRPF). There are different modes of uRPF. The uRPF strict mode allows inbound packets only if the incoming interface is the best reverse path, which doesn't work with asymmetric routes. The uRPF feasible mode accepts the packet if the inbound interface is on one of N feasible reverse paths. IDPF [31] constructs inter-domain packet filters based on feasible paths inferred from BGP updates, assuming that export policies of different commercial relationships are well conformed to. But the observed pervasive valley route announcements [70] and the recent survey of routing policies [39] indicate that those export policies are not always enforced, thus affecting the correctness and effectiveness of IDPF.

Accountable IP Protocol (AIP) [7] uses a hierarchy of self-certifying addresses. For source accountability, AIP uses ingress filtering as well. Besides trusting traffic from trusted neighbors (by uRPF) transitively, AIP also allows routers to verify the source address with the source address verification protocol, when the source AS of the packet is not a trusted neighbor and doesn't pass uRPF. They argued that ingress filtering was marginally effective because it relied on correct operator action. AIP helps to make ingress filtering automatic with uRPF and the source address verification protocol in a way that requires no configuration or interaction by operators or end-users. In this dissertation, we aim to enhance routing security by customizable routing policies thus improving the effectiveness of uRPF-based ingress filtering by automatically constructing ingress filters based on accepted routes.

Packet authentication based approaches like Passport [58], ICING [65] and OPT [55], require the source host (or AS) to stamp each packet with one MAC for each AS (or router) on the path to the destination. At each hop, a router extracts its MAC from the packet header and verifies it using a secret key shared with the sender. Dataplane approaches offer strong source authentication and path validation, but they require updates to the existing routers, increase packet sizes, introduce validation overhead for each packet at each hop, and must generate and manage the shared keys by some method. Moreover, firewalls may block packets with extra header fields.

Our goal is to enable and demonstrate routing integrity, source accountability, and programmable security policy using a logical trust platform based on trust Datalog—Datalog with authenticated statements. It is a control-plane only approach that interoperates with standard SDN dataplanes.

# 3

# ExoPlex Controller

## 3.1 Introduction

Network testbeds like ExoGENI [13], Chameleon [62] and ESnet [1] have been advancing with programmable infrastructures and network stitching capabilities (see §3.2), allowing us to deploy and evaluate heterogeneous NSPs on the testbeds. My research develops abstractions and tools to build such NSPs with logical security and policy control. We propose the ExoPlex controller framework for running virtual NSPs with testbed-hosted slices that generalize to testbeds with these enabling capabilities[1]:

- **Dynamic slices with virtual dataplanes.** Network testbeds provide APIs to provision network topologies and program them with SDN. In our model, participant slices act as *NSPs* that offer transit service for IP traffic through their networks.[2]
- **NSP peering.** Testbed slices may declare *stitchports* (see §3.2.2) and interconnect (stitch) them by mutual consent [90], e.g., at an exchange site or by allocated circuits. Testbed support for cross-slice stitching enables NSP slices to peer at L2 programmatically, even if they have different owners.

[1] In this dissertation, we run ExoPlex on research testbeds including ExoGENI, Chameleon and ESnet. But we can extend ExoPlex for other testbeds or public clouds with the enabling capabilities.

[2] We program NSP dataplanes with OpenFlow SDN, which is limited to IP.

- **Customer opt-in.** A slice may peer with an NSP provider and exchange IP traffic over the link. In addition, campus SDN networks increasingly support bypass services that enable authorized subnets to route/accept selected traffic through a dynamic L2 network circuit. In this way a network owner can “opt in” as a customer of a testbed-hosted NSP, effectively configuring it as an alternate Internet Service Provider for selected IP prefixes.

The experimental NSP services in testbeds can also carry real user traffic across research fabrics. For example, we envision that NSPs can offer security-managed connectivity with policy controls to enable or disable flows; impose security scanning or other NFV service chains on specified flows; protect against spoofing, hijacking, and DDoS attacks; or configure other defenses that are lacking in the public Internet. We work to support inter-domain traffic control within a network of NSPs, which may be experimental (e.g., user-managed), elastic, dynamic, and/or restricted to certain classes of traffic, e.g., high-priority data for a specific project. In the long term, we want to enable advanced network services as NSPs that weave into the fabric of the Internet over time through cycles of innovation and adoption.

In this chapter, I first describe the foundations on ExoGENI, Chameleon and ESnet on which we build ExoPlex controller—Ahab and stitchports—that our collaborators at RENCI implemented as part of our collaborations on this project. Then I introduce the ExoPlex controller architecture for running customizable NSPs in testbeds. As an exemplary NSP, we develop virtual SDX and demonstrate the potential for elastic NSPs with add-on values like NFV and QoS on testbeds with proof-of-concept experiments. This chapter is reprinted from [90, 91, 92] with permission.

## 3.2 Slice control and Cross-Slice Stitching

The slice abstraction of GENI [17] and its predecessors offers a natural container for *multi-cloud* applications that span multiple cloud sites and/or providers in a distributed cloud. A slice is a named set of virtual resources—such as VMs and L2 network links—allocated from infrastructure providers in a networked multi-cloud. A virtual resource that is managed

independently of other resources is called a *sliver*. Slices are the granularity of access control for GENI slivers: a user with permission to control a given slice can control all of its slivers. Therefore, we can view a slice as a *virtual network domain* controlled by its owners.

In the GENI multi-cloud, slice owners (tenants) can create virtual network topologies at the data-link layer (L2) within their slices. Compute nodes within a slice communicate over the *slice dataplane*, which is a linked private network that may span multiple sites within the multi-cloud. The cross-site links are typically bandwidth-provisioned L2 network circuits obtained from providers outside of GENI, e.g., the Internet2 AL2S and ESnet research network fabrics. These circuits enable high-speed slice dataplanes across sites, and experimentation at higher layers of the protocol stack including L3. However, communication between GENI slices has been restricted to interaction at L3, over the public Internet, as in the earlier PlanetLab model. Our goal is to support direct cross-slice networking at L2: it is more flexible to experimenters and the links are bandwidth-provisioned, so they are less vulnerable to contention from competing traffic.

To enable slices to connect to one another at L2, our collaborators at RENCI implemented new features for dynamic *cross-slice stitching* in ExoGENI [13, 27, 10], one of the two GENI "racks" deployments. The cross-slice stitch is effectively an L2 network peering point between two slice networks. If the slices are owned by different tenants then the stitch requires mutual consent.

They also worked to enable network stitching for Chameleon [62] and ESnet [1]. Chameleon is a testbed system that provides bare metal access to its two large hardware pools at the University of Chicago (UC) and Texas Advanced Computing Center (TACC). And ESnet is a scientific research infrastructure that provides high-speed network links. ExoGENI, Chameleon and ESnet allow users to create a network topology for an SDN-enabled IP dataplane. We can run Open vSwitch [68] in ExoGENI slices to support SDN, and Chameleon and ESnet provide hardware support for SDN with Corsa switch virtual forwarding context(VFC). We can also stitch slices across different testbeds including ExoGENI, Chameleon and ESnet and stitch slices in testbeds to slices in the public cloud like AWS with

L2 network circuits [76].

Cross-slice stitching enables testbeds like ExoGENI, Chameleon and ESnet to serve as an extensible platform for *inter-domain* networking across slices controlled by different tenants. Tenants may cooperate to build and maintain inter-domain networks in the testbeds. For example, slices with high-speed network circuits may function as *virtual autonomous systems* as a basis for new approaches to inter-domain routing. More generally, cross-slice stitching generalizes the PlanetLab idea of *unbundled management* [67]—in which user slices extend the testbed platform by providing useful network-based services to other slices—to encompass flexible, elastic L2 inter-networking.

This section reports on our exercises in inter-domain networking based on ExoGENI slice peering and slice peering across ExoGENI, Chameleon and ESnet. We first outline ExoGENI's new features for cross-slice stitching. We then report on an exemplary use case: a *network transit service* that runs within a slice, manages a shared circuit topology on behalf of a dynamic set of customer slices, and routes customer traffic over its internal dataplane (see Figure 3.1). This sharing of links in this exemplary shared transit service can mitigate the limited availability of high-speed network circuits, an important "pain point" in GENI today. Last, we report an exercise on providing network transit service with high-speed network in ESnet to tenant networks in Chameleon and ExoGENI.

### *3.2.1 Background: Networking on ExoGENI*

**GENI and ExoGENI.** ExoGENI is a networked IaaS testbed within the GENI federation. ExoGENI itself is a federation of xCAT/OpenStack cloud clusters spanning 20 locations across 4 countries on 3 continents, linked by the Internet2 AL2S and ESnet circuit fabrics. These infrastructure providers are called *aggregates* in GENI. ExoGENI control software supports end-to-end automated stitching of each L2 slice dataplane across multiple ExoGENI aggregates, bridging among circuit providers at ExoGENI exchange points (e.g., at Starlight) as needed.

**Managing slice-based network topologies.** Like other operations on virtual net-

works in ExoGENI, cross-slice stitching is dynamic and programmatically controlled. Our approach builds on earlier API extensions and internal mechanisms to allow tenants to modify existing slices and slivers dynamically [86]. In contrast, in other parts of GENI (e.g., the InstaGENI racks) a user creates all resources of a slice at each aggregate in one shot, and cannot modify a slice's footprint at any aggregate once the slice is instantiated there.

In this way, ExoGENI offers flexible support for tenants to build slices with multiple points of presence on multiple ExoGENI sites (e.g., for edge services), and to link them with a bandwidth-provisioned network backbone (dataplane) that is built to order for the needs of the slice. The modify operations enable the tenant to adapt its dataplane topology as its needs change. A slice has OpenFlow SDN control over its own L2 dataplane and inter-site traffic flow over its circuit links, so network policies governing packet flow are also under the control of the slice; the default policy is a single L2 domain spanning the slice dataplane.

**Semantic resource models.** ExoGENI is able to offer this flexibility in part through its use of a logic language (NDL-OWL [83, 12]) to describe slice requests, instantiated slivers, and sliver relationships. The control software uses the language to make statements about slivers. A semantic resource *model* is a set of statements about resources in the logic language. The ExoGENI control software (based on ORCA [27]) includes plugin extensions to orchestrate and provision slices by operating on these models. Slivers have globally unique names, so it is easy for a model to represent relationships among slivers in different sites or domains. The statements in the model are distinct and separable, and relevant relationships are encoded in the statements themselves rather than in the structure of the description document (e.g., as it is in GENI's XML-based RSpec). As a result, it is easy to change a model by adding or removing statements, or to combine models (e.g., across sites or domains) by taking their union. These properties enable the ExoGENI control software to modify slivers and slices dynamically and to reason about inter-domain infrastructure, e.g., to orchestrate end-to-end stitching automatically.

**Controllers and the Ahab API.** While semantic resource models have proven to be

powerful, it is difficult for users (and their programs) to produce and understand them. Our colleagues at RENCI introduced a library—Ahab—to hide logic concerns behind a new provisioning API to construct and modify slices. Tenants may implement long-running Ahab *controller* programs to manage their slices. A controller uses the API to construct, query, and modify the slice. For example, a controller can specify a sequence of changes to a slice and then *commit* those changes to launch the provisioning engine and inject requests into ExoGENI via a selected Slice Manager (SM) server. An ExoGENI SM determines how to map requests onto available aggregates, and orchestrates the execution of those requests in concert with their Aggregate Manager (AM) servers. The slice controllers for our transit slice experiment use the new Ahab API (§3.2.3) to build and interconnect the transit and customer slices (§3.2.4). The Ahab controllers also send commands to nodes within the slice over SSH.

The role of Ahab is similar to *geni-lib* [14], which provides a programmatic API for GENI requests. While *geni-lib* is interoperable with ExoGENI, it is limited to functions that are available using GENI-standard protocols and RSpec resource descriptions. Ahab goes beyond *geni-lib* in that Ahab enables sequenced changes to existing resources, including dynamic stitching to existing stitching points. The functions described here for dynamic slices, topology adaptation, and cross-slice stitching are not yet available through standard GENI protocols. Ahab accesses these functions using auxiliary protocols in ExoGENI in concert with NDL-OWL semantic resource descriptions and ORCA protocols. Note, however, that GENI slices can stitch to one another within ExoGENI even if they incorporate other resources outside of ExoGENI.

### *3.2.2 Slice Stitching in ExoGENI*

**Stitching and stitchports.** The *stitch* operation is a foundational building block for building dynamic L2 virtual networks. A controller program builds a slice by requesting various slivers and stitching them together. Each stitching point connects a pair of compatible slivers, e.g., a virtual node and a virtual link that are topologically adjacent at

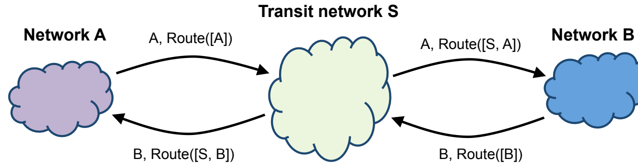



FIGURE 3.1: **An L3 network transit service in a slice. Customer slices attach to the service with cross-slice stitching and advertise IP prefixes. The transit network T is able to carry traffic for networks A and B, multiplexed over network circuits in T's slice dataplane.**

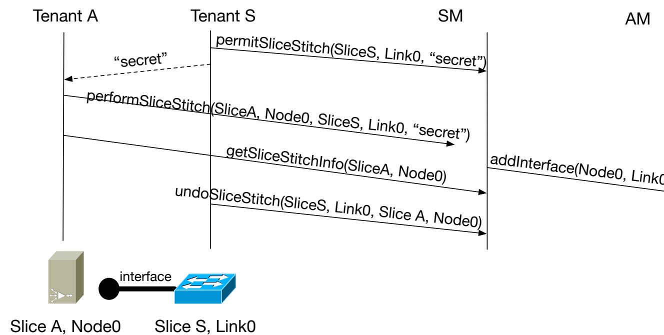



FIGURE 3.2: **Example message flow for slice-to-slice stitching. Tenant *S* creates a stitchable L2 link0 in its slice and passes a reference and secret token to tenant *A*, which uses the cross-slice stitching API to attach to it.**

some aggregate. In a cross-slice stitch, one slice exposes a virtual node or link as a type of named *stitchport*, so that another slice with suitable authority can stitch to it, creating an L2 peering point between the two slices.

A stitchport is a named meeting point where two independent network domains can be stitched together, establishing a L2 connection. ExoGENI has provided "static" stitchports for a number of years to connect ExoGENI virtual tenant networks (slices) to science assets outside of GENI [50]. A static stitchport is registered with an ExoGENI metadata service as a long-term network-facing endpoint attached to a fixed network segment (VLAN) behind the stitchport. We extend the stitchport concept with support for *dynamic virtual stitchports* exposed (or withdrawn) programmatically by the slices themselves. Controllers may use Ahab APIs to create and expose a slice stitchport, or stitch to another slice's stitchport (§3.2.2).

Stitching between ExoGENI slices is prototyped as new set of API calls with simple parameters that drive the underlying stitching functionality. In this implementation, any node-like or link-like sliver—a virtual machine, a bare-metal node, a point-to-point or a multi-point link between nodes—can become a stitching point. A stitch is only possible between a node-like and a link-like sliver, and the slivers must be hosted on the same aggregate and provider substrate/site. Currently at most one stitch is allowable between any pair of slivers.

In principle, it is possible to implement stitch requests for slivers on different aggregates or slivers of like types. For example, the cross-aggregate case might instantiate new link slivers to extend the specified link to the site that hosts the specified node. Any link-to-link stitching involves establishing a translation of VLAN tags (or other labels) for the stitched links, in order to join them together. These extensions are future work: the current API supports only local stitches between one link and one node.

To instantiate a local cross-slice stitch, the host aggregate's AM merely creates a new network interface on the specified node and attaches it to the specified link. The basic add interface/remove interface *sliverModify* operations were implemented earlier as part of ExoGENI's generic *sliceModify()* machinery [86]. Once the L2 stitch is established, an existing ExoGENI-specific extension ("neuca") [13] within the node's OS recognizes the new interface and configures it up (e.g., with *ifconfig*). Our experiments use node-to-link stitching, which allows the requester (the node owner) to pass an IP address for the new interface as a parameter.

Rather than explicit advertising of long-lived link stitchports, our prototype for dynamic stitching relies on bearer-token based authorization of stitching operations between slices, with one tenant (tenant $S$) allowing stitching operations to a particular sliver (e.g., a link serving as a dynamic stitchport) to any peer slice who knows a specified secret token. $S$ then communicates the token out of band to tenant $A$. $A$ then requests a stitch, passing the token as proof of access.

The slice stitching prototype adds five user-facing API calls to the ExoGENI Slice

Manager (SM) servers. This API may be invoked by command-line scripts or by ExoGENI's GUI tools and client libraries (e.g., Ahab):

- **permitSliceStitch**(<sliceID>, <sliverID>, <secret>). Establish a bearer-token secret for a stitchable sliver in a slice. One active secret per sliver is currently permitted. Internally a secret is stored as a salted hash of the secret and stored as a property of the sliver.
- **revokeSliceStitch**(<sliceID>, <sliverID>). Revoke any secret for a stitchable sliver, blocking future requests to stitch to it. Existing stitches are unaffected.
- **performSliceStitch**(<from sliceID>, <from sliverID>, <to sliceID>, <to sliverID>, <secret>, <stitch properties>). Request a cross-slice stitch between two compatible slivers. After checking that the slivers are colocated at the same AM, the SM invokes the AM to install the stitch. The stitch properties may include IP address, bandwidth attributes, etc.
- **undoSliceStitch**(<from sliceID>, <from sliverID >, <to sliceID>, <to sliver- ID>). Break an existing L2 stitch. Either side may break the stitch unilaterally. This serves as an "emergency break", e.g. in the event of a cross-slice network attack.
- **getSliceStitchInfo**(<sliceID>, <sliverID>). Request information about stit- ching state and events on a sliver owned by the caller. Reports whether this sliver allows stitches (a secret is set) and its stitching history: start and end times, identities of slices, slivers and their owners. Currently active stitches are determined by absence of end-time for the reported stitch. Each stitch operation is given a unique GUID.

Our Ahab controllers for the transit slice experiments use these API calls, as shown in Figure 3.2.

### *3.2.3 Instantiating and Connecting Slices Programmatically*

Many GENI users are familiar with using ExoGENI through the GENI tools based on standard RSpec (e.g. Jacks [48] or geni-lib). The GENI API and its tools are crucial for experiments that require slices spanning multiple testbeds in the GENI federation. However,

the advanced functions of ExoGENI—including automated dynamic stitching and cross-slice stitching—are available only through its native API.

Client programs may call ExoGENI's native API directly, but these API calls produce and consume logical/semantic resource descriptions in NDL-OWL (§3.2.1). Initially, the only way for a user to invoke the native API without confronting NDL-OWL was to use ExoGENI's Flukes GUI to draw and visualize slice topologies. This has long been a hurdle to using ExoGENI for complex experiments that demand more automation or that cannot be drawn easily in Flukes.

We introduce the Ahab Java library, which allows users to create and control ExoGENI slices programmatically while accessing all of the functionality provided by the native API. The Ahab API enables developers to create, modify, and destroy slices and resources on ExoGENI using simple Java objects. It is easy to incorporate it into a Java program: it is available from a Nexus Maven repository as a Maven dependency. With Ahab, it is now possible to create large complex slices easily and to create controller applications that can monitor, reason about, and modify long-running slices based on arbitrary policies.

The remainder of this section discusses the Ahab slice/sliver abstractions and the tools and workflow used to create and manage slices with Ahab. It offers simple Java classes and objects to abstract ExoGENI slices and resources. The primary object types used by an Ahab controller application are:

- **Slice**. Represents a slice.
- **Resource Objects**. A set of types representing each type of sliver available: ComputeNode, Network, StorageNode, Stitchport. These objects hold the configuration information for slivers, e.g. a ComputeNode's boot script or a Network's bandwidth. They also serve as an access point for accessing manifest information, e.g. a ComputeNode's management IP address or the IDs of the VLANs that compose a Network.
- **Interface**. The interface between two stitched resources. An interface contains information specific to that type of stitch. For example, an Interface between a Network and a ComputeNode might contain an IP address.

- **SliceProxy**. Manages the user's credentials required to submit requests to an ExoGENI Slice Manager (SM) server. Creating a SliceProxy requires a user's GENI certificate and private key, as well as the url of a preferred ExoGENI SM to use.
- **SliceContext**. Manages the user names and public SSH keys that are to be installed in nodes within a slice.

The following is an example of the workflow that is used to instantiate a simple slice (some of the code is simplified for brevity):

```
1    proxy = getSliceProxy(cert,key,url);
2    context.addToken(userName,pubKey);
3    s = Slice.create(proxy, context, name);
4    ComputeNode n = s.addComputeNode("n0");
5    n.setImage(imageURL,imageHash,imageName);
6    n.setNodeType(nodeType);
7    Network net = s.addBroadcastLink("net0");
8    Interface if0 = net.stitch(n);
9    if0.setIpAddress("172.16.0.1");
10   if0.setNetMask("255.255.255.0");
11   s.commit();
12   s.wait();
13   String ip = n.getManagementIP();
```

This code builds a SliceProxy and SliceContext in lines 1 and 2. These objects manage the certificates and keys to interact with the new slice and the ExoGENI provisioning services. Line 3 shows how to initialize a Slice object using the SliceProxy and SliceContext. No resources are provisioned for the slice until the code requests a sequence of updates to the empty slice (lines 4-10) and then commits them (line 11). In this example, line 4 adds a ComputeNode called "n0". Lines 5 and 6 set the node's image and node type. Line 7 addes a network. Line 8 stitches the node to the network creating an Interface. Lines 9 and 10 set the IP and netmask of the Interface. The commit action at line 11 forms the NDL request and sends it to the ExoGENI Slice Manager (SM) specified in the proxy.

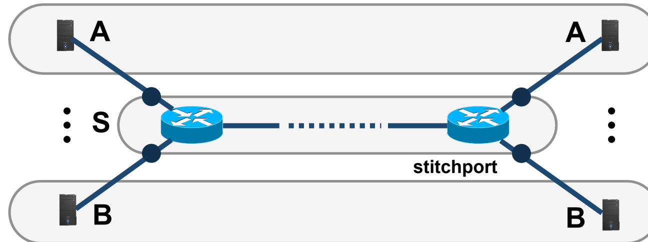



Figure 3.3: **A dumbbell topology for evaluating the network transit and exchange service for GENI slices. The nodes in the carrier slice are linked by a shared VLAN (broadcast link) that spans the sites over a single I2/AL2S dynamic network circuit. The carrier slice transits traffic for its customer slices, which (in this example) have no connectivity among their PoPs except through the carrier slice.**

The commit returns as soon as the SM accepts the request, but the steps to provision the slice and its slivers proceed asynchronously. The Ahab controller may wait() (as in line 12) until the pending updates to the slice are complete (or failed). When wait() returns, the Ahab controller might perform additional configuration and/or send requests into the slice to kick off an experiment. In this example, the ComputeNode object is used to access manifest information that is necessary to complete the configuration and run an experiment: line 13 shows the application obtaining the management IP address of the ComputeNode. It can then use this IP to copy executables and other files to the ComputeNode and invoke any necessary tasks, e.g., with *ssh* and *scp*, authenticated by the keys in the SliceContext.

### *3.2.4 Demonstration Experiments*

**Example: Network Transit Slice.** As a running example, we consider a simple *transit slice* ($S$) that routes network traffic over its dataplane on behalf of customer slices that attach to it (e.g., $A$ and $B$). In this example, the intent is to carry a customer's traffic from one of the customer's points-of-presence to another, and not necessarily to allow communication between the customers. This example has a compelling motivation: it frees $A$ and $B$ from the need to request cross-site network circuits for their own dataplanes, conserving the scarce supply of network circuits available to GENI. The transit service is suitable for GENI slices that want high-speed dataplane connectivity but are willing to multiplex their

traffic over circuits shared with other transit customers.

To use the transit slice, a customer must conform to the transit provider's L3 network model (if any) to allow traffic routing. We experiment with two simple alternatives for the transit provider: (1) a virtual L2 transit service that bridges disjoint L2 networks in a customer slice, and (2) an L3 domain that uses OSPF internally to route traffic among non-conflicting IPv4 prefixes advertised to it by its customers.

We conducted simple demonstration experiments for ExoGENI cross-slice stitching: we deployed two variants of a slice-based *network transit service* that passes customer traffic over its slice dataplane, i.e., it acts as a *carrier* slice. An Ahab controller for the transit slice runs as a service (outside of the slice itself), and accepts requests from customers to stitch to the transit slice at particular PoPs—cloud sites that host nodes and/or links for both slices. For each request it returns a sliverID and token that the customer slice may stitch to (see Figure 3.2). Each customer slice is controlled by its own Ahab program, which requests stitches from the transit controller.

The transit service provides connectivity to customer slices through its peering points. For example, tenant slices $A$ and $B$ may interconnect their PoPs through the transit slice, as an alternative to allocating dedicated network circuits for their own dataplanes. Figure 3.3 depicts an example using a simple dumbbell topology for a transit slice occupying two PoPs—the configuration we use in all of our experiments. The PoPs for all slices are instantiated on ExoGENI sites at two universities in Florida: UNF and UNL.

Once a cross-slice stitch is in place, either peer may inject arbitrary traffic into the other slice across the slice boundary. As with any inter-domain networking, each slice controls how it directs traffic into the peering point and how it handles incoming traffic. We experimented with two approaches for the transit slice:

- **L3/OSPF.** The transit slice and all customer slices run a standard IPv4 stack on Linux VMs (Ubuntu 14.04) enabled for Quagga/OSPF and configured with non-conflicting IP prefixes (RFC 1918 private addresses). When a cross-slice link is instantiated, each peer automatically advertises its routes to the other.

- **L2/SDN.** The nodes in the transit slice run Open vSwitch (OVS) with a shared SDN controller—an extended Ryu SimpleSwitch controller called PriorityNet- work—that implements a single L2 domain. PriorityNetwork can segregate traffic from the different customers (e.g., by swallowing ARPs and blocking broadcasts) and schedule traffic from different customers according to assigned priority weights. It ignores packet header fields above L2, and so preserves GENI's flexibility for network experimentation at L3 and above.

The Ahab controllers take minutes to instantiate a virtual network for a slice: the delay is limited primarily by the time for the ExoGENI rack sites to launch VMs on their underlying OpenStack/Linux/KVM clusters. We used I2/AL2S network circuits for the transit slice: requests for such links are faster to provision, but they are often denied due to resource constraints.

Once the nodes and links are active, a cross-slice stitch takes 10 seconds or less. In fact, the stitch is instantiated almost immediately—the Ahab wait() primitive reports that the stitch is active—but the neuca extension within a node polls at 10-second intervals to discover each new interface and configure it up. The customer's Ahab controller uses *ssh* (via the Jsch Java library) to notify the slice when the stitch request commits and wait() returns.

To illustrate, the Ahab customer controller for the L3/OSPF example works as follows. It uses *scp* (Jsch) to copy a stitch configurator script onto each node in its slice as it comes up. When a newly requested stitch is complete (wait() returns), the controller invokes the configurator program via *ssh* on the peering node within the slice, passing the configured IP address. The configurator loops until *ifconfig* reports an interface with the expected IP address. It then writes the interface data into Quagga's configuration file and kicks Quagga to reload its configuration, which triggers OSPF route advertisements across the new interface. We found that it takes about 50s for Quagga to propagate new routes across all nodes.

Our measurements suggest that traversing a stitch has effectively zero performance cost.

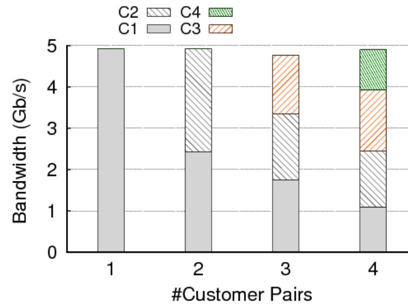



FIGURE 3.4: **A shared link in a carrier slice is shared fairly among flows from multiple customer sender/receiver pairs. All flows go through a 5Gb/s bottleneck network circuit that links the dumbbell sites. The number of node pairs is increased from 1 to 4. Each sender/receiver pair establishes 10 connections.**

Single TCP streams (*iperf* with default parameters) between VMs on the same aggregate reliably yield 9.2-9.3 Gb/s over the 10 Gb/s Ethernet interconnect used for ExoGENI dataplanes, regardless of whether the nodes reside in the same slice or the traffic traverses a cross-slice stitch. CPU utilization is below 70% on the sender and below 85% on the receiver (single-core VM, XO-medium). This result substantiates our view that inter-domain networking at L2 is a useful vehicle for GENI experiments involving data-intensive and high-speed network services that run in slices. Moreover, slice-based services accessed by cross-slice stitching are reasonably secure: connectivity is "off by default" and is enabled only by mutual consent, and the peer is strongly authenticated at least by its sliceID.

Figure 3.4 reports results of an experiment in which multiple attached customers pass traffic between their PoPs through an L3/OSPF transit slice with single shared 5 Gb/s I2/AL2S network circuit connecting the PoPs. Customer flows in this example use TCP. As expected, the TCP congestion algorithm naturally shares the link fairly among the flows. As discussed in §3.2.1, this simple demonstration example is not practical without additional authorization (e.g., to validate IP prefixes) and perhaps traffic policing in the transit/carrier slice.

The L2/SDN alternative illustrates the flexibility of network control within the carrier slice. In this example, the nodes in the carrier slice control network traffic with an OpenFlow SDN controller that implements a "big learning switch" at L2. The SDN controller

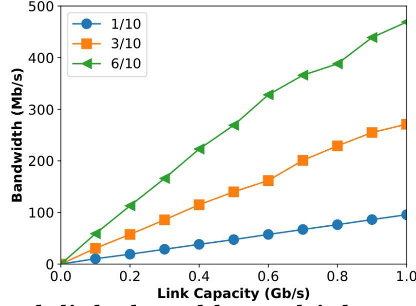



FIGURE 3.5: **A single bottleneck link shared by multiple customer sender/receiver pairs. We use OpenFlow queues to limit traffic rate between the pairs. Each customer gets bandwidth proportional to its share.**

is extended to use OpenFlow QoS queues to provide differentiated service to different customers. We use different OpenFlow queues for traffic between different pairs, with different relative shares 1, 3 and 6. We measured the bandwidth each pair gets as link capacities of the carrier slice change. As shown in Figure 4, the bandwidth each pair gets is in proportion to its configured share weight.

### *3.2.5 Network Stitching across Research Testbeds*

Virtual networks in different network testbeds including ExoGENI, ESnet and Chameleon can be stitched with L2 circuits provisioned from a circuit provider with connections to multiple testbeds, such as Internet2/AL2S. We can stitch networks between Chameleon and ExoGENI by creating a stitchable network on Chameleon and adding a static stitchport with the same VLAN tag to the ExoGENI slice [2]. We can also connect Chameleon networks at TACC and Chameleon networks at UC with circuits via ExoGENI by stitching two Chameleon networks at different sites to an ExoGENI slice with two static stitchports. Stitching networks in ESnet to Chameleon networks and ExoGENI networks can be performed by privileged administrators.

As a demonstration, we run ExoPlex (§3.3) on an ESnet VFC network, operator-configured for this purpose, to provide network transit service to client networks at Chameleon and ExoGENI. We created three client networks on Chameleon at UC, Chameleon at TACC, and ExoGENI at RENCI. Our collaborators at RENCI helped creating the VFC

network that contains three VFCs on different sites and high-speed links connecting the VFCs, and stitched the client networks to the ESnet VFC network at three sites. The three client networks can communicate with each other via the ExoPlex-based network transit service on the ESnet VFC network.

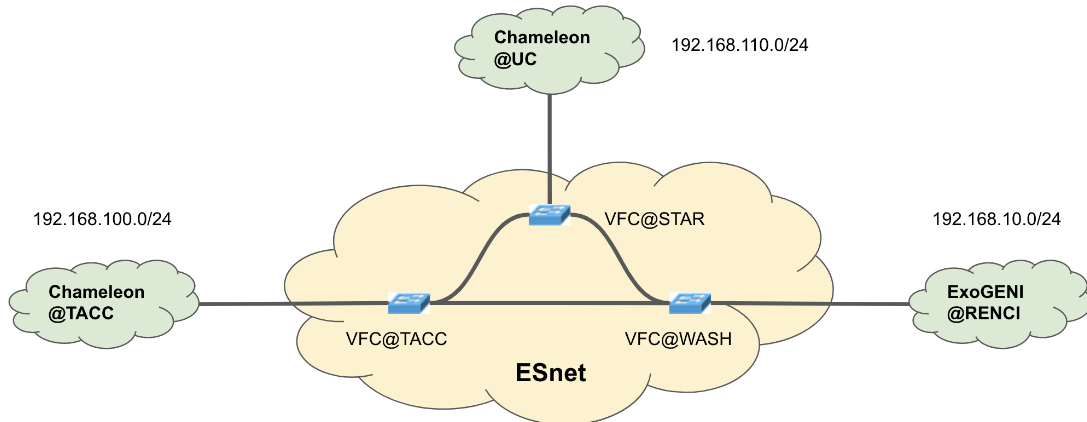



FIGURE 3.6: **A demonstration experiment: connecting client networks at Chameleon and ExoGENI with a VFC network in ESnet.**

The network stitching capability across testbeds allows us to integrate resources from different testbeds and harness the power and strengths of different testbeds. For example, one can deploy datacenter in Chameleon cloud for computation-intensive work, deploy edge networks in ExoGENI for better locality and connect the datacenter and edge networks with express backbone network from ESnet. And the hardware support for SDN in Chameleon and ESnet makes it possible to deploy production-grade networks in the testbeds.

## 3.3 ExoPlex

With the enabling capabilities, we want to support experimentation with NSPs and safe interconnection of NSPs on network testbeds. We propose the ExoPlex controller framework. ExoPlex-based NSPs can provide network services with QoS, NFV and secure policy-based multidomain networking.

ExoPlex defines a standard interface for cross-domain interactions involving NSPs and/or

their customers. An NSP controller is a server, operated on behalf of an NSP, that speaks for the NSP and commands its network. The controllers interact with one another and with other servers operated on behalf of their customers. The ExoPlex control plane operates at the level of the per-domain NSP controllers.

Additionally, an NSP controller commands the NSP's dataplane—its network of SDN switches—through the northbound APIs of its SDN controller(s). It may also call virtual hosting APIs (e.g., Ahab) to add or remove virtual switches or links. We assume a separate control network for all of these control-plane interactions. The control network is operated by infrastructure providers—campuses, cloud providers, and network testbeds—and is accessed via the public Internet.

The NSP controllers export RPC APIs (e.g., REST/HTTP) to control inter-domain peering and networking. Table 3.1 summarizes selected northbound control plane APIs for an NSP controller. The authenticated customers and peers invoke these APIs to attach (stitch) an L2 link and to enable specified IP traffic to flow over the link. These calls propagate the routes and policies that govern traffic flow, which are encoded in logical certificates.

ExoPlex also provides APIs for operators to manage NSPs. An NSP operator can call those APIs to initialize peering requests to peer NSPs or originate a route/policy advertisement. Those APIs enables automated/scripted experiments and tests with multiple involving NSPs (see §5.4). Those functions are not the core of ExoPlex and can be implemented independently.

### *3.3.1 Software Architecture*

Figure 3.7 depicts an ExoPlex NSP and its controller, which is layered above its SDN controller(s), the SAFE logical trust engine, and a testbed-specific IaaS plugin (slice controller). The ExoPlex control plane operates at the level of the per-domain NSP controllers. An NSP controller is a server, operated on behalf of an NSP, that speaks for the NSP and commands its network. The controllers interact with one another and with other servers

**Table 3.1: Control plane APIs of an NSP controller. Customers and peers invoke these REST APIs to attach (stitch) a node to the NSP, to notify it of routes for a peering link and of policy rules governing the use of those routes, and to request for bandwidth provisioned connection to edge NSP/SDX.**

| | |
|---|---|
| **stitchRequest**(slice ID, sliver ID, secret, stitch properties) | Stitch a sliver (node) in a customer or peer slice to an NSP edge node at the same site. ExoGENI supports such cross-slice L2 stitches guarded by a *secret*, as in [90]. |
| **undoStitch**(slice ID, sliver ID) | Discard a stitched L2 link between the peer/customer sliver and the NSP slice. |
| **stitchportRequest**( stitchportURL, vlan, stitch properties) | Stitch a science network outside of GENI to the NSP slice at a static stitchport [90]. |
| **advertiseRoute**( route, route cert) | Advertise a route, with a link to the signed certificate of the route. |
| **advertisePolicy**( src, dst, policy cert) | Advertise a path control policy for traffic from the source prefix to the destination prefix, with a link to the signed policy certificate. |
| **connectionRequest**( subnet A, subnet B, bw) | Request for bandwidth-provisioned connection between subnet A and subnet B. |

operated on behalf of their customers. Additionally, an NSP controller commands the NSP's dataplane—its network of SDN switches—through the northbound APIs of its SDN controller(s). It may also call virtual hosting APIs to add or remove virtual switches or links. We assume a separate control network for all of these control-plane interactions. The control network is operated by infrastructure providers—campuses, cloud providers, and network testbeds—and is accessed via the public Internet.

**SDN-enabled IP Dataplane** ExoPlex works on SDN-enabled IP dataplane. It manages traffic forwarding in a network of OpenFlow-enabled switches, which is independent from the IaaS platform.

**Slice Controller.** The slice controller is a IaaS platform specific plugin that manages the slice given the capabilities of the platform. For ExoGENI, the IaaS plugin uses the Ahab library to build and maintain the NSP's topology by invoking ExoGENI's API for dynamic slices(§3.2.3). ExoPlex extends to testbeds other than ExoGENI. NSPs may replace the IaaS

plugin for another dynamic slice API, or run without one over any static SDN-programmable dataplane topology. We have deployed ExoPlex NSPs over Corsa switch VFCs (virtual forwarding contexts) in the Chameleon and ESnet testbeds.

**SDN Controller.** ExoPlex includes OpenFlow SDN controller software to program the NSP dataplane, based on an extended Ryu *rest-router* module for routing and mirroring traffic, an extended Ryu *rest-qos* module for QoS, and Ryu *ofctl-rest* module when necessary.

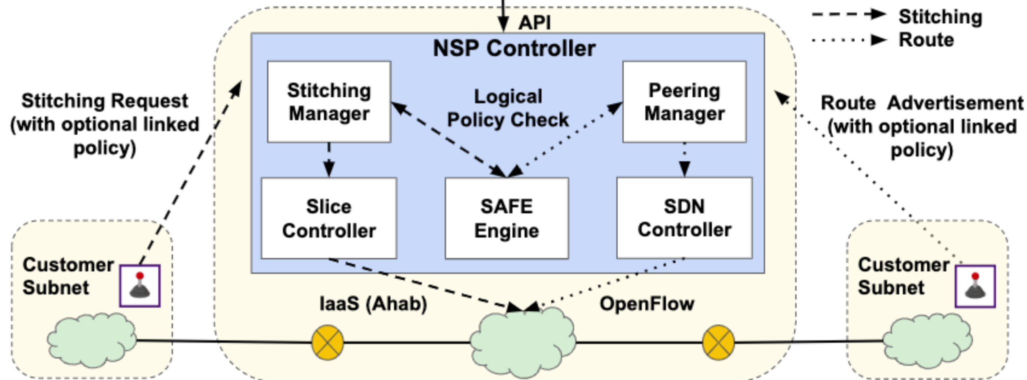



FIGURE 3.7: **An exemplary ExoPlex Network Service Provider (NSP). The NSP dataplane comprises network circuits—allocated from a circuit provider such as I2-AL2S—that link one or more sites or points-of-presence (PoPs); each site runs optional NFV appliances and OpenVSwitch routers controlled via OpenFlow. The NSP controller is a server program that invokes: testbed APIs to orchestrate the slice; OpenFlow controller(s) to manage traffic flow within the slice; and a local SAFE engine to produce, consume, and validate logical certificates and check compliance with logical policy rules. The NSP controller exposes a northbound API for permissioned peering, permissioned flows, and policy-based path control.**

An NSP controller exposes northbound control plane APIs for its customers and peers to request peering links and notify the NSP of new policies and routes. Calls to these APIs drive all control plane interactions to propagate routes and policies across the interdomain network. The handler for an incoming call invokes a local SAFE engine to perform various validation checks, then optionally modifies its network state and propagates notifications to peers. Outgoing route advertisements are signed under the NSP's keypair. NSP controllers are assumed to be reachable to one another, e.g., on the public Internet.

ExoPlex includes a standard set of controller API handlers and SAFE trust scripts, which together determine when and how to install or withdraw routes and filtering rules in

the NSP dataplane via the SDN controller APIs. We extend the SDN controller for *ingress filtering* and *source-specific routes* to support policies for path control and anti-spoofing defenses. The trust scripts define logic templates, standard validation rules for incoming routes; hooks for custom authorization rules for peer requests and permissioned flows [91]; and custom policy rules to filter outgoing routes. We extended these rules to validate multi-hop paths through multiple transit NSPs.

## 3.4 Virtual SDX: a Prototype NSP

The advancing network testbeds expose the capabilities of advanced research fabrics to IP-based edge subnets (opt-in via campus), built-to-order IP networks to serve science communities. Data-intensive science collaborations increasingly provision dedicated network circuits to share and exchange datasets securely at high speed, leveraging national-footprint research fabrics such as ESnet or I2/AL2S. The network stitching capability in research network testbeds supports automatic circuit interconnection of science resources across campuses and in network cloud testbeds, such as GENI (e.g., ExoGENI) and NSF Cloud (e.g., Chameleon). Taken together, these tools can enable science teams to deploy secure bandwidth-provisioned virtual science network (virtual SDX) that links multiple campuses and/or virtual testbed slices and provides network transit services with integrated in-network processing on virtual cloud servers.

We built a dynamic virtual SDX with security monitoring with ExoPlex as a prototype tenant NSP. The SDX concept was conceived as a physical facility (PoP) with direct attachment to customer networks [42]. IaaS providers with broad reach, such as the ExoGENI federation, can host virtual SDX services that provide similar functions decoupled from fixed peering points. A virtual SDX can extend to many geographically distributed PoPs via a dynamic bandwidth-provisioned network backplane topology.

The virtual SDX is an elastic slice managed by an Ahab controller. The controller runs outside of the virtual SDX slice and exposes a northbound API for operations on the slice by peer domains that are authorized to peer with the virtual SDX as customers. Customers

invoke these northbound APIs to bind named subnets under their control to the virtual SDX via L2 stitching, request bandwidth-provisioned connectivity with other subnets (including subnets owned by other customers), and specify logical policies that govern approval of requests by other customers to connect to them.

The virtual SDX slice comprises virtual compute nodes running OpenVSwitch, OpenFlow controllers, and Bro traffic monitors. Traffic flow and routing within the virtual SDX slice are governed by a variant of the Ryu *rest-router* SDN controller similar to the *Plexus* SDN controller used within Duke's campus SDN network. The virtual SDX slice controller computes routes internally for traffic transiting the virtual SDX network, and invokes the northbound SDN controller API to install them. The SDN controller runs another Ryu module (*rest-ofctl*) to block traffic from offending senders. If a Bro node detects that traffic violates a Bro policy, it blocks the sender's traffic by invoking a *rest-ofctl* API via the Bro NetControl plugin.

There are two ways for customers to request for connectivity to virtual SDX: (1) explicit bandwidth-provisioned connectivity request via virtual SDX's northbound API; (2) implicit connectivity request without bandwidth reservation by sending a first IP packet to the peer subnet.

At customer request for bandwidth-provisioned connectivity, virtual SDX calls the slice controller to provision slice resources as needed to carry the expected traffic. These resources include peering stitchport interfaces at each PoP, the OVS nodes that host these virtual SDX edge interfaces, Bro nodes to monitor the traffic, and backplane links to carry the traffic among the PoPs. The controller reuses existing resources in the slice if they have sufficient idle capacity to carry the newly provisioned traffic, and instantiates new resources as needed. In particular, it adapts the virtual SDX backplane topology by allocating and releasing dynamic network circuits as needed to meet its bandwidth assurances to its customers.

The virtual SDX installs rules in OVS flow tables that forward the first packet of a new connection between a subnet pair to the SDN controller. Then the SDN controller caches the packet in a buffer and calls the virtual SDX controller's northbound API to request for

connection. The virtual SDX controller authorizes the connection between the subnet pair that the source and destination IP address of the packet belong to. If the connection is authorized, virtual SDX controller calls SDN controller to set up routing path between the subnet pair, and send out the buffered first packet for the connection. To protect virtual SDX controller from being overloaded with connection requests triggered by packets from unauthorized connections, the SDN controller issues the connection request for a packet only if it hasn't received any packet in the past 4 seconds between the source and destination IP address.

I demonstrate elastic traffic monitoring service and elastic transit service for virtual SDX in §3.4.1 and §3.4.2 respectively. And I discuss authorized network stitching and connectivity in §4.3.

### *3.4.1 Elastic Traffic Monitoring Service*

NSPs can provide NFVs for tenant traffic. As an exemplary NFV, we inspect permitted flows with out-of-band Bro network security monitor applicances to detect intrusion in our virtual SDX. I leave the discussions about logical trust for slice stitching and connectivity for Chapter 4. As a simple form of intrusion prevention, it uses Bro's *NetControl* framework to interrupt all traffic from the source of a suspect flow. The virtual SDX controller deploys Bro instances elastically to scale capacity as customers join. Bohatei [33] provides an elastic DDoS defense by deploying filtering appliances in a similar way. We assume that the virtual SDX controller knows the volume of tenant traffic from their requests and forward the tenant traffic within some flow space to Bro nodes with enough capacities. Thus we don't consider state migration for Bro when scaling the pool. OpenNF [38] is a controller architecture that manages internal NF state and controls network forwarding with SDN. S6 [85] supports elastic stateful network functions by using distributed shared object (DSO) to manage states.

Figure 3.8 shows the NFV controller that manages elastic scaling of traffic monitoring service in virtual SDX. To support fast out-of-band traffic monitoring, we need an SDN

controller to mirror the traffic to security appliances while forwarding traffic to its destination. I extended ryu *rest-router* module for this goal. To support flexible mirroring rules that may be independent from routing rules, I used separate OpenFlow tables for mirroring and routing in OVS. In table 0, I install traffic mirroring flow entries to mirror the required traffic to Bro, while sending all packets to table 1, where the packets will be processed and forwarded with routing flow entries.

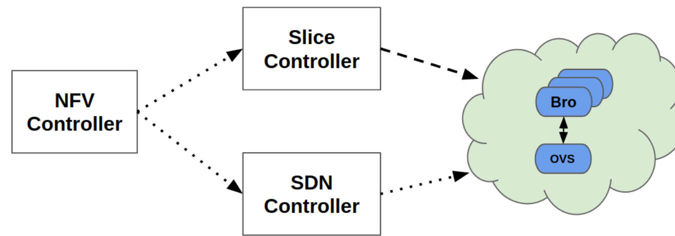



FIGURE 3.8: **NFV controller in ExoPlex that scales Bro pool via slice controller and set traffic mirroring rules via SDN controller.**

Bro is a powerful, open-source out-of-band network monitoring and analysis framework supporting a wide range of traffic analysis tasks, including network activity logging, sanity checks for various protocols, and intrusion activity matching. It has been widely used by network infrastructures and service providers.

Typically, a Bro instance is deployed alongside an edge router, where intrusion detection is needed. A router or switch mirrors network traffic to the out-of-band Bro instance, which scans the network traffic without affecting performance. Scanning applies a script of match-action rules to identify and flag suspicious activities. If a sampled traffic pattern matches a rule, Bro triggers its event engine to take a corresponding action. These actions are selected from a library of connectors (event handlers) in Bro's *NetControl* framework. Typically, these actions add the sender's IP address (or subnet) to a blacklist and/or use SDN to cut flows or sandbox endpoints.

Bro may also be deployed in a closed loop to install rules that block (blackhole) traffic from a suspected attacker on ingress at the network edge. Bro tooling may also cross-reference with NetFlow data and install rules in access-filtering edge devices to block attack

traffic.

I explore the performance of closed-loop traffic control based on Bro when operating within a virtual SDX slice on the ExoPlex platform. Our experiments use synthetic traffic, but we source and sink the traffic from a dynamic set of customer slices that stitch to the virtual SDX at L2, modeling a live deployment. Due to limitations of current testbeds, we are limited to virtual host-based network appliances and SDN-based access control within the virtual SDX slice. These functions run within OpenVSwitch (OVS) instances. The Bro and OVS instances run on elastically deployed ExoGENI VMs. An Ahab slice controller is responsible for elastic provisioning of the monitoring and filtering capacity within the virtual SDX slice.

*Bro Evaluation*

In this section, we evaluate the effectiveness of the virtual SDX Bro nodes under varying traffic. This evaluation explores both the effectiveness and limitations of deploying Bro in the ExoGENI testbed, and stability (reproducibility) of the results across multiple deployments of the same declarative slice specification.

Our virtual SDX NSP experiment consists of an ExoPlex slice deployed across two ExoGENI sites, which interconnects four client domains (see Figure 3.9). Two of the client domains are located on the Chameleon testbed, and connect to the virtual SDX using dynamic circuits and stitchports. The other two client domains are independent slices on ExoGENI, each connecting to the virtual SDX using slice-to-slice stitching at each PoP—the virtual SDX establishes a PoP on each ExoGENI site where it has an authorized customer. For simplicity, all client domains specify SAFE policies that allow incoming traffic from all other client domains, but require that the secure ingress service terminates flows identified as potentially malicious according to a pre-established set of Bro match-action rules.

Each link within the ExoPlex virtual SDX is allocated 2 Gbps of bandwidth, and the link between ExoPlex and each client domain is 1 Gbps. All ExoGENI nodes (clients and ExoPlex services) are VMs having 4 cores and 12 GB RAM (the “XO Extra Large” instance

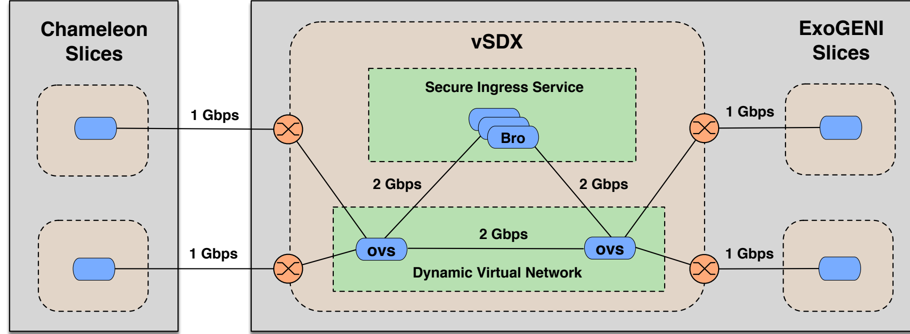


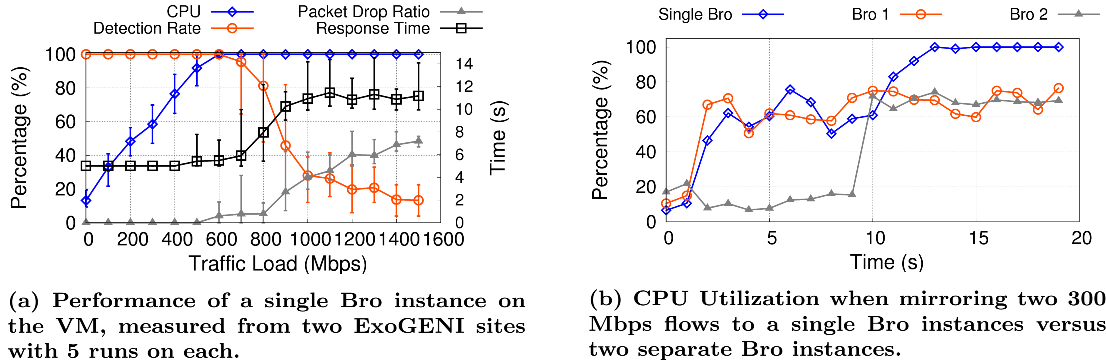

FIGURE 3.9: **An experimental SDX network that exchanges traffic between customer nodes and mirrors traffic to Bro nodes for intrusion detection.**

type). Two types of nodes comprise the virtual SDX: OVS nodes (running version 2.0.2) and Bro nodes (running version 2.5.2). On the Bro nodes, we load all policy scripts included in the distribution, and add one of our own that detects transfers of "malicious" files with specific signatures. Should a "malicious" file be detected, Bro instructs the SDN controller to drop traffic from the source using the *NetControl* framework as described above.

We measured the performance of Bro nodes deployed on several different sites within the virtual SDX. In each run a Bro node filters a traffic flow between a pair of clients interconnected via the virtual SDX. The flows are synthetic and have similar traffic profiles, including attack traffic. For each experiment we recorded selected metrics defined as follows:

- **Response Time**: We define response time as the period between detection of a "malicious" file transmission and the termination of the associated connection by the SDN controller. Thus, we are able to quantify the delay in protecting clients from an attack.
- **CPU Utilization**: As processing demand increases with the traffic flow at a given instance, each instance's monitoring ability is eventually saturated.[3]
- **Packet Drop Ratio**: With high load of CPU utilization and mirrored traffic, Bro begins dropping packets. We define the packet drop ratio as the percentage of dropped packets to the total that should have been mirrored.

[3] Since each Bro instance is single-threaded it can saturate at most one core, so we clip the CPU utilization to 100% in the graphs. It may reach a peak of 110% under a high flow volume due to other activity on the VM.




**(a) Performance of a single Bro instance on the VM, measured from two ExoGENI sites with 5 runs on each.**

**(b) CPU Utilization when mirroring two 300 Mbps flows to a single Bro instances versus two separate Bro instances.**

FIGURE 3.10: **Bro performance in ExoPlex experiments.**

- **Detection Rate**: As packets are dropped, the likelihood of Bro failing to identify an attack increases. We define detection rate as the percentage of malicious files detected by Bro, relative to the total that are present.

During each run of our experiments, pairs of clients (one each from Chameleon and ExoGENI) send measured amounts of UDP traffic through the virtual SDX using `iperf3` [47]. All traffic traversing the virtual SDX is mirrored to a Bro node. While this sample traffic is flowing, FTP is used to transfer 200 "malicious" files that should be detected by Bro. Each run reports the four selected metrics.

A Bro instance can only process a bounded traffic load before dropping packets—about 600 Mbps under our settings (§3.4.1). To handle higher traffic loads, the virtual SDX dynamically launches multiple Bro instances and balances flow processing across them based on measured flow rates and customer requirements. When the used capacity of a Bro pool exceeds a certain threshold (i.e., 60%)—or no Bro instance at the specific edge PoP has sufficient remaining capacity to process new flows— the controller launches a new Bro instance. To demonstrate the effectiveness, we have two pairs of connections, with each of them sending traffic at the same rate (300 Mbps). The initiation of one of these two flows is offset by 10 seconds, during a given run of this experiment. We evaluated Bro filtering performance against our metrics, both when the traffic was mirrored to a single Bro instance and when the mirrored traffic was diverted to different Bro instances on a per-

client-pair basis. We do not have support for our OVS virtual routers to hash flows from the same (sender, destination) pair across multiple Bro nodes. For this purpose some switch vendors (e.g., Arista) have introduced high-speed switches with configurable *tap aggregation* to mirror traffic and distribute monitored flows across a cluster.

Figure 3.10a shows the results measured from multiple runs on different host ExoGENI PoP sites. At ∼600 Mbps of mirrored traffic, a single core begins to saturate: Bro begins dropping packets and the detection rate decreases. When backgroud traffic is less than 600 Mbps, the response time is constant (∼5 seconds), due to Bro's event scheduling mechanism. Beyond 600 Mbps of mirrored traffic, Bro's response time increases rapidly, but the detection rate does not fall until it begins dropping packets. As we expected, even a minor increase in packet packet drop ratio (∼18% at 900 Mbps) can result in a significant decrease of the detection rate (∼55%), since a few consecutive dropped packets disrupt matching for file detection. For predictable and reliable performance, a single Bro instance's processing capacity is limited to ∼500 Mbps (given the Bro scripts, traffic features, and VM types used in our experiment). The error bars indicate the performance variability we experienced on the ExoGENI VMs hosting Bro at different times and across different sites. We believe that performance is sufficiently stable to support elastic performance control at some cost in efficient utilization to allow headroom for variable performance.

Figure 3.10b reveals the effectiveness of scaling across multiple Bro instances for reducing the CPU utilization of individual instances. When we mirror both 300 Mbps flows to a single Bro instance, CPU saturation occurs after the second flow is initiated, 10 seconds into the run. When the individual flows are mirrored to separate Bro instances, both instances exhibit stable and predictable performance that can be attributed to the decreased CPU utilization by each individual instance.

These initial experiments are steps toward effective elastic configuration policies for scalable monitoring in the virtual SDX service. These policies require a performance model for Bro that predicts performance and effectiveness as a function of load and capacity. We can infer such a model and apply it for elastic provisioning in the virtual SDX slice

controller.

### *3.4.2 Elastic Network Transit Service*

An NSP can provide elastic network transit service with quality of service by dynamically adapting the topology of the service slice to meet the changing customer demands. We provision virtual service networks from infrastructure providers to provide network services to tenant networks. As tenant demand changes, or the physical network changes due to network failures or deployment of new infrastructure resources, we should dynamically adapt the virtual service network to serve tenant networks better and more efficiently.

I extended ryu *rest-qos* and *rest-router* to support bandwidth provisioned connections in dataplane. We use the QoS queue in OVS that supports traffic throttling by limiting the minimum or maximum bandwidth. To allow flexible traffic throttling rules that are independent from mirroring and routing rules, I implemented them in different OVS tables: ExoPlex uses table 0 for mirroring flow entries, table 1 for traffic throttling entries and table 2 for routing flow entries.

#### *Demonstration Experiment*

We conducted an demonstration experiment to provide network transit service with bandwidth requirements to tenant networks on ExoGENI. As ExoGENI doesn't support capacity adjustment on active links, we can add or remove links when necessary as the bandwidth requirement between two sites changes. Accordingly, we modified Ryu *rest-router* module so that it works correctly with multiple links between the same routers: (1) We modified the ARP learning process to avoid flooding ARP requests via the loop; (2) We added mapping between the port and MAC addresses, so ExoPlex can use the right link with required bandwidth to meet the customer demands.

On ExoGENI with the I2-AL2S circuit provider, it takes about 1 minute to provision a new cross-site link. When a connection request can not be fulfilled with existing links, ExoPlex provision new links with 1.5$x$ capacity as required to allow for faster processing

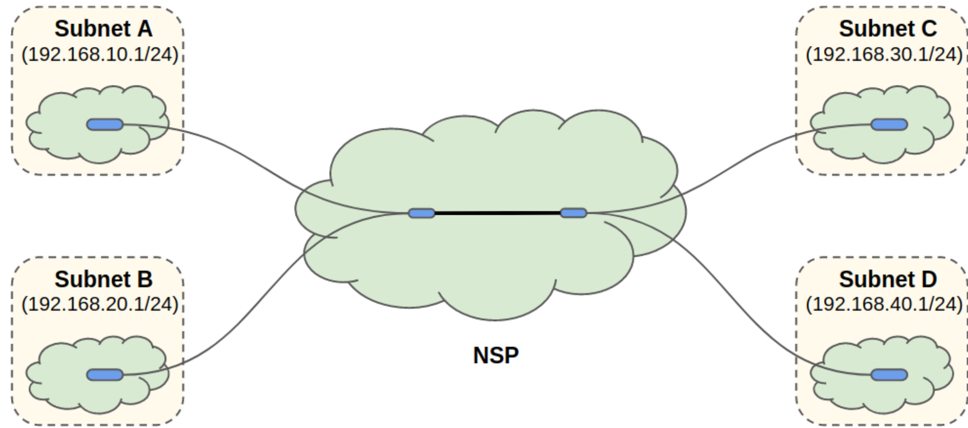



FIGURE 3.11: **An NSP provides network transit service with bandwidth requirement to tenant networks. The NSP dynamically add new links between the two sites to meet the increasing bandwidth requirements between subnets.**

of future demands. I throttled the bandwidth between subnets with Ryu rest-qos Module at the ingress edge of the NSP. Figure 3.11 shows the network topology of four tenant networks and an NSP network. There are two pairs of tenant networks on different sites. After stitching the four tenant subnets to the NSP slice, different subnet pairs required for connections with different bandwidths one by one. At each conneciton request, the NSP provisioned a new link between the two sites if there was no available path with required bandwidth between the subnets. Table 3.2 shows the bandwidth requirements between subnet pairs, provisioned link capacity by the NSP, and the measured bandwith between subnet pairs with `iperf` [47]. At the end, the NSP provisioned 3 new links to meet the bandwidth demands between subnet pairs.

**Table 3.2: Required, provisioned and measured bandwidth between tenant subnets.**

| Subnet Pair | Required Bandwidth | Provisioned Capacity | Measured Bandwidth |
|---|---|---|---|
| A and C | 100 Mbps | 150 Mbps | 93.4 Mbps |
| B and D | 300 Mbps | 450 Mbps | 278 Mbps |
| A and D | 100 Mbps | 0 | 93.8 Mbps |
| B and C | 200 Mbps | 300 Mbps | 185 Mbps |

## 3.5 Summary

This chapter details the underlying technology for NSPs, NSP peering, and the ExoPlex controller architecture for NSPs. The demonstrations show how a controller can assemble dataplane, add with embedded security processing such as virtual Bro intrusion NFV appliances, and adapt the dataplane and NFV deployment elastically according to demand.

# 4

# Logical Trust for Multidomain Networking

## 4.1 Introduction

A decade ago the research community launched major initiatives to combine network testbeds to leverage benefits of scale, diversity, geographic dispersion, and heterogeneity. NSF GENI [18, 19] in the US and FIRE in the EU exemplify this trend. Both initiatives have funded deployment of IaaS federations spanning many sites and providers. The cross-testbed network stitching capability enables *multi-cloud* applications that span multiple network testbeds. We can use multiple providers and combines the strengths of different testbeds. For example, we can use Chameleon for computation intensive applications like security monitoring, ESnet for high-speed networks, and ExoGENI with geographically distributed sites for high-speed networks and edge computing. NSPs on those network testbeds can provide network service to a common community of users, which may require some form of federated identity for their users (a *community cloud* [64]).

The three dimensions of network federation—peering, multi-cloud, and community—raise a variety of challenges for managing identity, resource access, naming, object access control, network stitching and connectivity. It requires some means to represent the trust and certify trust relationships among users and providers, including their terms of peering.

It also places new pressure on the mechanisms to manage the security of the multidomain interactions like stitching, connectivity and routing.

This chapter shows how to address those challenges with a logic-based approach comprehensively and uniformly. First, I introduce SAFE [22] trust management system (§4.2). Our collaborators built SAFE and I made some enhancements to the logic engine and evaluated its performance. Then, I introduce template logic rules for common services like group service, resource allocation and delegation. After that, I introduce basics of logical trust in ExoPlex (§4.3). As an example, I discuss logical trust for network stitching and connectivity that are fundamental for virtual SDX (§4.4). Last, I evaluate the performance of logical trust for virtual SDX and demonstrate authorized stitching and connectivity with an experimental virtual SDX and tenant subnets deployed in ExoGENI (§4.5). This chapter is reprinted from [22, 24, 91] with permission.

## 4.2 Foundations: SAFE Logical Trust

SAFE [22] is a trust management system that use logic statements to express facts, policies, endorsement and delegations. It has a logic engine (Styla [82]) to reason about a query, a distributed key-value store to store, share and protect credentials, a logic-based scripting language called "slang" to integrate the trust logic with applications and insulate them from logic concerns, and a slang interpreter. We can use SAFE for trust management and authorizations in network and federated clouds.

Our collaborators built SAFE before I joined the project. I made some enhancements to the logic engine and participated in its performance evaluation.

### *4.2.1 Overview*

**Building with trust logic.** SAFE's trust logic is based on Datalog [26], a rigorously defined and extensively studied general-purpose logic language that is a subset of Prolog, a popular language for logic programming with a standard syntax. It adds a modal operator **says** to Datalog, enabling its direct use as a logic of belief and attribution, following

Binder [30], SD3 [51], and SENDLOG [5].

Datalog content consists of atomic statements (atoms) and rules built up from atoms and the logical operators conjunction and implication. An atom is a predicate symbol applied to a list of parameters, which may be variables or term constants representing principals, objects, or values. Predicate symbols are user-defined: they may represent properties, attributes, roles, relationships, rights, powers, or permissions. Atoms whose parameters are term constants (ground) represent simple assertions equivalent to a row of a database table. Rules embody implication and may contain variables. A rule has a head and a body. The head of a rule is a single atom. The body is a sequence of atoms (*goals*) separated by commas, which indicate conjunction: all of the atoms in the body must be true for the rule to "fire". A rule allows the prover to infer that the head is true for some substitution of its variables with constants, if the body is true under that substitution.

In Datalog-with-says, every atom has a first (prefix) parameter representing a principal who **says** it (the *speaker*). If the parameter is omitted, it defaults to the current principal (`$Self`). In this way, a statement about a principal naturally represents a delegation or endorsement that is restricted by the speaker and predicate; another principal considers the statement only if it has a policy rule with a matching goal, conferring trust in the speaker.

Datalog-with-says is sufficiently powerful to represent common access control features hierarchical naming, nested groups, roles and other attribute assertions, ACLs, and capabilities. Delegations may be constrained by a predicate/role and by parameters (e.g., *"Alice owns slice S"*). Conjunctive policy rules permit reasoning from multiple attributes of a principal or object, and policies are mobile: they may be passed in certificates.

SAFE defines conventions for self-certifying term constants (IDs) to name principals and objects. A principalID is a SHA hash of the principal's public key, following SPKI/SDSI [32]. All statements in a valid certificate must have a speaker ID that matches the issuer who signed the certificate. Each object named in a logic statement has some principal who is its *controlling authority.* The objectID consists of an identifier (a UUID/GUID) chosen by its authority, concatenated with the authority's principalID to form a *self-certifying identifier*

*(scid)*. SAFE scripts use a builtin function `rootPrincipal` to obtain a scid's controlling principalID. Self-certifying IDs ensure that parties have distinct names for their objects, and a malicious principal cannot "hijack" another's names. In this way logical trust extends conventional identity-based PKI security to incorporate rich statements about principals, objects and their security attributes, and avoids the need for a global naming root.

**SAFE logic scripting and certificate linking.** SAFE synthesizes elements from previous trust logic systems and extends them with additional system support to enable practical deployment. The novel elements of SAFE include a scripting language to insulate applications from logic concerns, and an interface to a shared key-value store (e.g., a DHT), which stores authenticated logic content as signed certificates in a native SAFE format. Certificates in the store are indexed by self-certifying links (*tokens*), and can be written only by their issuers. The application trust scripts contain parameterized logic templates to generate certificates easily, and also to link certificates to construct DAGs programmatically as a side effect of delegations.

The use of *certificate linking* simplifies discovery and retrieval of the content relevant to a trust decision. The certificate links (tokens) also enable pass-by-reference and caching of certificate content at the authorizers. The shared certificate store enables an issuer to update or revoke its certificates by their tokens, addressing common PKI concerns.

Scripting is organized around the abstraction of *logic sets*—sets of logic statements that represent credentials, delegations, endorsements, and policies. Scripts use templated constructors (`defcon`) to construct and modify sets and link them to form unions.

A principal may issue (*post*) its logic sets and share them by reference; posted sets are materialized as certificates spoken by the issuer and signed under its keypair. A posted set is accessible to any client that knows its token, but only its issuer can modify it. Scripts name their locally constructed sets with arbitrary string names (*labels*); the token is a SHA hash of the issuer ID and the label. Thus tokens are "unguessable", but anyone who knows the label and the issuer's public key can synthesize a set's token. Some script actions (e.g., name resolution) obtain links in this way.

SAFE *guard* scripts (`defguard`) combine linked sets to construct query contexts, and issue queries to check policy compliance for trust decisions. SAFE fetches a certificate when a guard references a logic set by its token. After validation SAFE extracts the semantic content of the certificate into a logic set cached in an in-memory *set cache*. The scripts deal only with the semantic content: the SAFE runtime encodes and decodes logic material, handles cryptographic operations, and performs fetch, retrieval, and caching automatically and transparently.

We assume that all participants run scripts with common logic/certificate templates, although they may install different policy rule sets. (This assumption assures interoperability, but it is not required for security.) Each participant runs its SAFE server with their own scripts, which are all under its direct control: authorization is naturally end-to-end [45].

**Certificate repository.** SAFE's certificate store is suitable for decentralized operation with the trust properties of a permissioned blockchain deployment, but with a more scalable implementation. Specifically, it is intended to run as a Byzantine quorum system (BQS) following Phalanx [61]. These systems scale more easily than blockchains because they allow sharding, in which each operation executes on only a subset of replicas. They are sufficient for logical trust because the logic programming model does not depend on a linear sequence of operations (state-machine consensus) as imposed by blockchains, in which all operations execute on all replicas in a strict linear order. Thus "unchained logic" offers a scalable alternative to blockchains as a foundation for decentralized trust. In this thesis, I use an enterprise key-value store (Riak [73]) when applicable.

**Memoization in logic engine.** Styla uses a top-down approach for logic reasoning. Reasoning about a query in Styla is similar to expanding an And-Or tree. The queried goal is the root of the tree. Matching and unifying a goal with a logic rule generates subsidiary goals in an "and" branch: the goal is true if all subgoals in the "and" branch are true. Matching a goal with different logic rules/facts results in different "or" branches: the goal is true if the subgoals in any "or" branch are true. Styla uses a depth-first search strategy when reasoning. There could be multiple inference paths leading to the same subgoal.

Styla unnecessarily repeats the proof for the same subgoal, and the cost of time could be exponential to the depth of the And-Or tree. When a predicate without variable or without bound variable (i.e, any variable in the predicate doesn't appear in other subgoals of the "and" branch) is reasoned to be true or false, we can memoize the result for the predicate. Once we reach the proved predicate via another reasoning path, we look up the result instead of proving it again. This is called memoization. We found memoization important when reasoning route import/export queries against the routing policies of NSPs (see §6.2.2), where an object can be delegated membership to a group via hundreds of delegation paths. I added memoization to Styla and this reduces the time complexity from exponential to polynomial for certain queries.

### *4.2.2 Off-the-Shelf Logic Rules*

SAFE comes with off-the-shelf logic rules and scripts that users can import and use for common services like group service, resource allocation, endorsement and delegation.

**Group service.** It is often useful for participants to assert their own attributes about one another. Any principal may declare a group as an object, and issue certificates granting ownership or membership in the group with named privileges or roles. Members may delegate their rights to others transitively using a capability model.

Listing 4.1 shows a standard set of logic rules to govern the delegation by checking endorsement chains.

**Listing 4.1: Logic rules to authorize if a user is a member of the group by checking endorsement chains.**

```
membership(?Group, ?User) :-
  membership(?Group, ?User, _).

membership(?Group, ?User, ?Delegatable) :-
  ?GRoot := rootPrincipal(?Group),
  ?GRoot: groupMember(?Group, ?User, ?Delegatable).

membership(?Group, ?User, ?Delegatable) :-
  ?Delegator: delegateMembership(?User, ?Group, ?Delegatable),
  membership(?Group, ?Delegator, true).

membership(?Group, ?User, ?Delegatable) :-
  ?GRoot := rootPrincipal(?Group),
  ?GRoot: nestGroup(?Group, ?ToGroup, true),
  membership(?ToGroup, ?User, ?Delegatable).
```

```
membership(?Group, ?User, ?Delegatable) :-
  ?GRoot := rootPrincipal(?Group),
  ?GRoot: nestGroup(?Group, ?ToGroup, false),
  ?ToGroupRoot := rootPrincipal(?ToGroup),
  ?ToGroupRoot: groupMember(?ToGroup, ?User, ?Delegatable).
```

The rule set is linked to the group, and may be customized, e.g., to manage specific roles or privileges. Listing 4.2 shows a constructor to create a group, and Listing 4.3 shows a simple constructor to grant membership in a group. When invoked with concrete IDs as parameters, these constructors return sets with logical assertions declaring the existence of the group, its owner and members, and its governing policy set. These sets may be posted as linked certificates, enabling other parties to query group memberships, e.g., to control access.

**Listing 4.2: Set/certificate constructor for a typed user-defined group object owned by a specified subject and linked to a set of policy rules controlling delegation of rights to this object.**

```
defcon createGroup(?SubjId, ?GroupId, ?Policy) :- {
  owner($SubjId, $GroupId).
  group($GroupId).
  link($Policy).
}.
```

**Listing 4.3: Set/certificate constructor to delegate membership in a group to a subject. A boolean indicates whether the receiver may delegate it further.**

```
defcon addMember(?GroupId, ?SubjId, ?Delegable) :- {
  groupMember($GroupId, $SubjId, $Delegable).
  link($GroupSetRef).
}.
```

**Route Validation.** Each route advertisement is represented by a logic certificate. Each hop of a route is a logical assertion advertising to a peer NSP a route for a specified destination prefix, along with an ordered list of predecessors (PrincipalIDs) in the path: `advertise(?DstPrefix, ?Path, ?Peer)`. The issuing NSP invokes a script to encode the advertisement in a logical certificate and sign it under the issuer's keypair. The certificate links to the next hop in the chain of predecessor advertisements.

Listing 4.4 shows the template script for a customer network to originate a route for its IP prefix. It links its customized path control policies and the certificate for its allocated IP prefix. Listing 4.5 shows the template script for an NSP to sign an advertised route.

To propagate a route, an NSP invokes this script to issue an `advertise` statement for each eligible peer, adding itself to the head of a list of NSPs that describes the route. It then invokes its peer's control API, passing the token to inform it of the route.

Each route advertisement links to the certificate for the previous hop. The links create chains of certificates, enabling standard logical rules to validate an entire path. List 4.6 shows the logic policy that an NSP enforces to verify a route advertisement.

**Listing 4.4: A template script for a customer network to originate a route advertisement with links to its customized routing policies and the certificate proving the allocated IP prefix of its network.**

```
defcon originateRoute(?DstIP,?Path,?Target,
  ?IPCert):-
    ?Policy := label("custom policy"),
    ?NspAcl := label("nsp-tag-acl"),
  {
    link($IPCert).
    link($Policy).
    link(NspAcl).
    advertise($DstIP,$Path,$Target).
  }.
```

**Listing 4.5: A template script for an NSP to sign a route advertisement with a link to the signed statements of the previous hop and a link to the certificate that proves the tags of its network.**

```
defcon advertiseRoute(?DstIP,?Path,?Target,
  ?Cert):-
    ?TagSubjectSet := label("tags"),
  {
    link($Cert).
    link($TagSubjectSet).
    advertise($DstIP,$Path,$Target).
  }.
```

**Listing 4.6: SAFE routing logic. The AS($Self$) verifies a received advertisement by authorizing the route advertisement chain from the prefix owner. ($Path$ is a list of ASes. $eq([?Head|?Tail], ?Path)$ is a built-in function that assigns the first element of $Path$ to $Head$ and the rest to $Tail$.**

```
authorizedRoute(?Owner, ?DstIP, ?Path, ?AS):-
  eq([?Owner|?Tail], ?Path),
  eq(?Tail, []),
  ?Owner: advertise(?DstIP, ?Path, ?AS),
  ownPrefix(?Owner, ?DstIP).

authorizedRoute(?Owner, ?DstIP, ?Path, ?AS):-
  eq([?Head|?Tail], ?Path),
  ?Head:advertise(?DstIP, ?Path, ?AS),
  authorizedRoute(?Owner, ?DstIP, ?Tail, ?Head).
```

**Prefix ownership.** The origin of a valid route must own the advertised destination prefix. The origin links its initial advertisement to a certificate set with evidence that it

owns the prefix. As the route propagates, each NSP in turn applies local policy rules to this logic set to validate the origin's ownership of the prefix. ExoPlex includes a trust script to delegate a prefix to a named principal, linked to a predecessor as evidence that the issuer owns the containing prefix.

Origin authentication ensures that the first advertisement of a prefix issues from a principal that is duly authorized to control routing to the prefix. The origin must be valid according to statements issued by authority principals according to some trust structure. The authority structure and logical checks ensure that endpoints communicate using non-conflicting IP prefixes and are prevented from stealing or controlling one another's traffic.

Our approach is analogous to the RFC 6480 architecture (RPKI), but implemented using SAFE. The profiles for resource certificates are given by a logical vocabulary within the standard SAFE certificate format, validated by the logical rules in Listing 4.8. We do not use special end-entity or Route Origination Authorization (ROA) certificates as RPKI does; instead, any owner of a prefix may originate a route for the prefix to a provider network (e.g., an edge NSP or SDX). The SAFE certificate store acts as the distributed repository system, but linked using SAFE's general hashed tokens. In contrast, RPKI organizes stored certificates in a hierarchy, which is restrictive but also allows filesystem-like naming.

The governance policy for prefix ownership identifies a set of one or more roots of authority for the address space, via local policy statements at each participating NSP that those principals are considered authoritative for specified prefixes and have the right to allocate sub-ranges from them. One option models current IP governance as reflected in RPKI deployments: the local policy of each participant states that a root namespace authority (e.g., IANA/ICANN and its Internet Registries) controls all IP address space and allocates sub-ranges (prefixes) to owning principals hierarchically and transitively. Participants must agree on the root authority and the form of the certified delegations, or else they fail to validate one another's prefixes. An alternative is to ground prefix ownership in a forest of *a priori* anchors for disjoint segments of the IP address space. This alternative is more practical for inter-domain networking on testbeds in that it does not rely on global authority

deployment.

Listing 4.7 shows the issuer script with the template for a resource certificate. The certificate contains a logic statement to `allocate` an IP prefix, i.e., to declare that a subject principal `$Holder` holds the prefix. It also links to another certificate for support: the token `$Cert` is the head of a chain of resource certificates grounded in some authority, and proving that the issuer owns an IP prefix containing the more specific `$Prefix` that it sub-allocates in this new resource certificate. Anyone with this certificate may invoke a guard that fetches the chained certificates and validates the prefix ownership and that the chain is grounded in some locally accepted authority. The guard validates by applying the logic rules in Listing 4.8. We added a builtin operator (`<:`) to the logic engine to validate containment of IPv4 prefixes specified in a standard string format.

**Listing 4.7: A template script to post a statement of IP prefix allocation to anther principal with a link to the certificate proving that the issuer owns the allocated IP prefix.**

```
defcon ipAllocate(?Holder,?Prefix,?Cert) :-
  {
    link($Cert).
    allocate(?Holder,$Prefix).
  }.
```

**Listing 4.8: The logical rules for prefix ownership authorization. A principal owns (holds) an IP prefix if the trust root or an upstream one who owns the prefix allocats the prefix to it. "<:" is a builtin operator in SAFE logic engine to determine if the former prefix is within range of the latter. The two arguments of the "<:" should have be unified to explicit prefixes beforehand to avoid unbounded logical inferences.**

```
ownPrefix(?Holder,?Prefix):-
  $TrustRoot: allocate(?Holder,?Prefix).

ownPrefix(?Holder,?Prefix):-
  ?UpStream: allocate(?Holder,?Prefix),
  ownPrefix(?UpStream,?SupPrefix),
  ?Prefix <: ?SupPrefix.
```

Figure 4.1 illustrates how the trust scripts for these applications link sets according to the delegation patterns.

### *4.2.3 SAFE Evaluation*

**Context pruning and indexing.** Certificate linking and delegation-driven pruning can improve the efficiency of logic inference. We show this effect for nested groups (Figure 4.2)

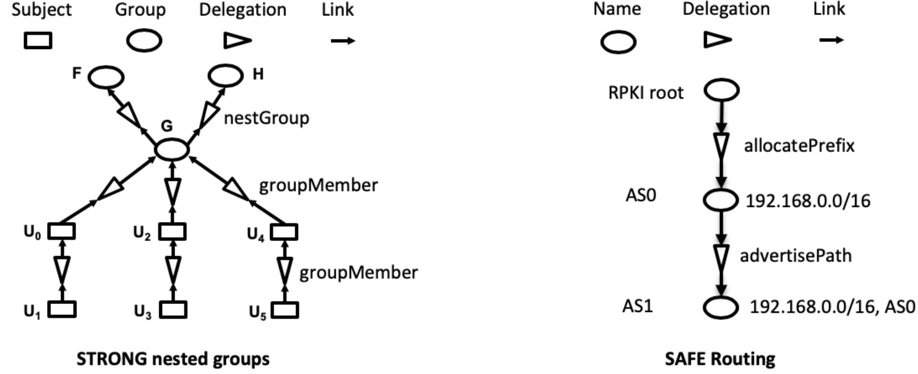



FIGURE 4.1: **Set linking examples. For nested groups, links traverse group delegation and membership delegation backwards; Secure routing links to IP prefix delegations and route advertisements to certify each route.**

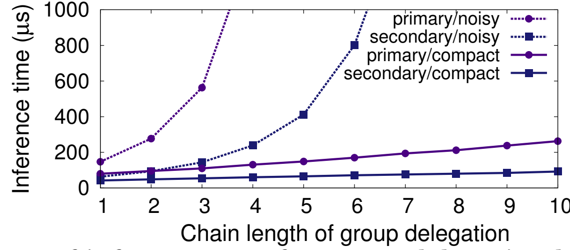



FIGURE 4.2: **Comparison of inference cost for group delegation between noisy and compact contexts, with primary and secondary indexing schemes. The baseline delegation chain contains 1 group membership and 5 membership delegations.**

in the group service. The structure of group delegations creates a hazard for a prover trying to answer an access query: *is principal $P$ a member of group $G$?* If membership in $G$ is delegated to multiple subgroups, recursively, then the prover may explore the subgroup closure searching for a subgroup that includes $P$. For the "compact" contexts these superfluous delegations are pruned, because they are not reachable in the link closure of principal $P$'s credential set for $G$. That is $P$ does not link any of its credential sets to them, because there is no associated delegation of interest to $P$.

We show the impact of linked pruning on the inference using noisy contexts with superfluous delegations. The linking patterns in the scripts prune these superfluous delegations, so a SAFE prover would not see them in real operation. In a noisy context, we inject the additional delegations as a binary tree and set the tree height to the length of the delegation

chain (given by the x-axis). I conducted experiments for subgroup delegation (Figure 4.2) in the group service.

The noisy contexts lead to exponential proof costs. In contrast, *the proof cost is always linear in the length of the delegation chain for the pruned context in conjunction with the secondary index.* In this group example, the prover explores it by applying group delegation rules backwards recursively from the root, and at each step it knows the containing group that it is looking for (initially it is $G$). The primary index indexes on the goal's predicate name and argument count, but the secondary index includes this first argument. It takes time linear in the number of facts to build the secondary index for a context, but once it is present the desired fact is extracted from the context in constant time.

**Inference cost for complex policies.** For these typical operations and scenarios in the cloud federation example, SAFE identifies and retrieves a tightly bounded superset of relevant certificates for each trust decision automatically, and the cost of compliance checks is linear with proof length. However, more complex policies may show higher costs, particularly for disjunctive policies (complex ACLs, cross-federation with multiple trust anchors). The multiple branches force the prover to search each branch looking for a proof. For example, a user request for access may search a long list of groups in an ACL, looking for one that includes the requester.

To illustrate this concern and focus on the cost of the logical reasoning itself, Figure 4.3 shows the logical inference cost for access checks against a list of groups in an ACL, as a function of the length of the ACL and the depth of delegation of the user's membership in a single group in the list. Costs grow with the number of disjunctions (ACL length), as well as the cost to traverse the group delegation chain to form the proof of access. This delegation cost is linear in SAFE due to the use of a secondary index.

Overall, the results suggest that logical inference is cheap in the common case given that the certificate linking structures constructed by the scripts focus the prover on relevant logic content, and prune out extraneous statements. Thus logical trust is cheap in the common case: cost grows with the complexity of the policies, but we pay only for the policy

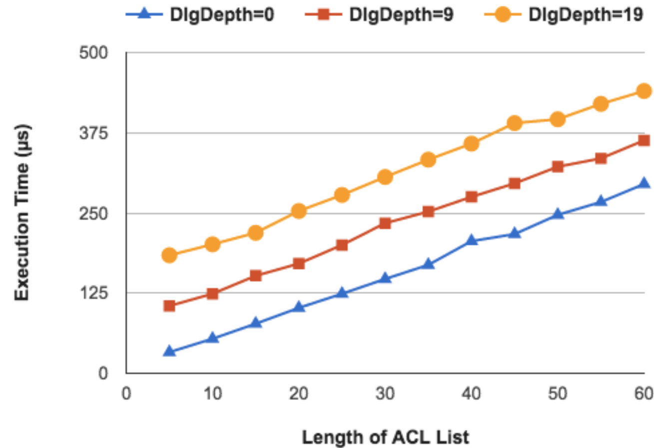



FIGURE 4.3: **Raw inference time for access checks against an ACL of groups, as a function of ACL length and group delegation depth. The logical inference costs are a few hundred microseconds for checks that are more complex than are likely to occur in practice.**

complexity that we use.

**Routing and IP delegation.** We examine the efficiency of queries in the **Routing** application under a synthetic scenario (Figure 4.4). We run a SAFE instance on a four-core KVM (Intel Xeon CPU E5520 @ 2.27GHz) with 12 GB of Ram and 1 Gb/s Ethernet. We run SAFE set Riak server on similar VMs with 75 GB of disk storage. AS principals exchange information on routing and IP prefix assignment/delegation in the same way as in BGP and RPKI. A guard checks whether an advertised path is valid by verifying route delegation at each hop and IP prefix ownership of the origin. We split the IP address space into 32K prefixes and delegate those prefixes from an (RPKI) trust anchor to 32K ASes. The branching factor of the delegation tree is 8 and the depth is 5. We generated a random network topology of those ASes, and ASes choose the shortest path to propagate routes. The average path length is 6.9.

Figure 4.4 shows the latency distribution for a single SAFE engine to bulk-validate all routes with the same origin, routes received by a given AS, and a random set of routes. This example illustrates a limitation of SAFE's linking and "pull-based" approach, which is not suitable in all cases. Pulling and caching help to validate random routes and for routes

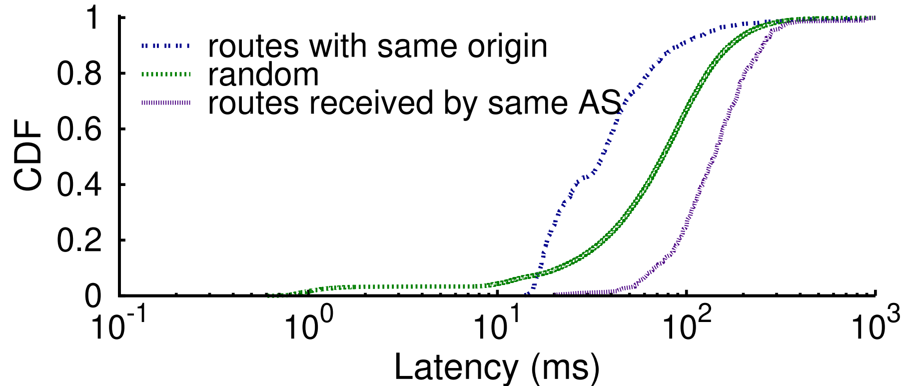



FIGURE 4.4: **Latency of routing verification under three query workload models: routes with the same origins, routes randomly selected, and routes received by the same ASes.**

with the same origin. However, in BGPsec each node (e.g., a router or host) must validate the routes that it receives, and each route advertisement is specific to the advertiser and the destination prefix. Thus a node considers each certificate exactly once, and its set cache yields no value. Moreover, "pulling" certificates on demand incurs fetch costs, while the conventional approach of "pushing" supporting certificates with each advertisement has the desired effect of leaving each node with exactly the certificates it needs, just when it needs them. Each certificate is needed by all successors in its path, but never by any other node. This is the worst case for SAFE's pull-based approach.

## 4.3 Logical Trust in ExoPlex

ExoPlex manages security data including security policies of related players with SAFE logical trust framework. It authenticates and enforce the security policies to make sure the network configurations are compliant to the security policies. The customization and flexibility of the security policies lead to flexible and programmable multidomain networking, making ExoPlex a powerful platform for policy-driven multidomain networking with a compact implementation.

**The players (principals).** Each participating network domain (NSP or edge subnet) is controlled by a security principal with a keypair. Interacting network domains are

necessarily embedded within some *governance* structure with additional principals, e.g., to assign addresses within a common space. For example, communicating subnets must own compatible IP prefixes delegated to them from common trust roots, and policies may rely on security tags (attributes) of principals or networks asserted by various endorsing authorities. Principals use their keypairs to sign their requests, delegations, policies, endorsements, and/or advertised routes.

**Governance.** ExoPlex supports an open governance model for flexible experimentation. Each party specifies the trust roots and governance rules that it subscribes to using logic. Parties may interact only to the extent that their structures and rules are compatible. The experiments in this dissertation use a simple governance model in which common trust anchors—accepted by all participants—delegate IP prefix ownership and endorse/certify NSPs with security attributes (tags). Exoplex builds its secure control network over the existing public Internet, e.g., so that NSP controllers can invoke one another's APIs for peering.

**Logical policy.** A *logical policy* is expressed as a set of logical facts and rules to govern and authorize multidomain interactions like peering, connectivity and routing. NSPs subscribe to standard rules to validate routes and authenticate IP origin prefixes. In addition, customer subnets may specify policies that guard connectivity to their prefixes and/or constrain the paths for inbound and/or outbound flows. Associated NSPs receive those policies and evaluate compliance. For example, a subnet's direct provider or virtual SDX (vSDX) receives connectivity policy from the subnet and blocks traffic from unauthorized senders on the last hop before delivery.

Policy rules may query statements and security attributes of other relevant parties. For example, connectivity rules may query attributes of the source. The policy also defines which authorities may assert/endorse these attributes. The standard route validation rules authenticate the origin as the owner of the prefix according to the NSP's governance rules.

The logical trust approach makes it easy to express and share governance policy in logic, independent of other elements of the implementation. A policy might express a federation

structure or, alternatively, a set of ad hoc trust agreements among the interacting parties. For example, the prefix ownership rules in our prototype express a structure similar to the public Internet, in which prefixes are delegated transitively through a hierarchy of owners, with range containment checked at each level. The participants must agree on the roots of authority, as in RPKI.

NSP controllers check policy compliance by issuing scripted queries to a local SAFE logic engine, passing a logic *context*—a set of certified facts and rules in datalog. The trust scripts construct each logic context and incorporate relevant assertions and policy rules extracted from signed SAFE certificates, and selected local logic.

The logic approach allows any participant to check compliance with another's policy on its behalf. For example, customers trust their edge providers (SDX) to enforce their connectivity policies.

## 4.4 Logical Trust for a Virtual SDX

A key difference from a classical SDX is that customer domains of virtual SDX specify policy using trust logic rather than through OpenFlow directly. The virtual SDX customers specify logical policy rules that govern which other customers may interconnect with them; these policies may consider the security properties of peer networks and/or the accountable identities that control them. Customers trust the virtual SDX to evaluate compliance of any connecting peers with their security policies on their behalf. Customers also present logical certificates that represent their own identities, subnet ownership, and security properties when they attach to the virtual SDX.

Logical trust provides a simple and powerful basis for authorization of requests to the virtual SDX service. The NSP APIs in Table 3.1 invoke guard scripts to check authorization before completing each call. Clients authenticate their REST calls to the virtual SDX API using keypairs. Clients pass links to certificate chains that express their connectivity policies and various attributes and permissions granted to them by other parties. These include certificates assigning identity attributes to their public keys, certificates from GENI

authorities endorsing the slice and naming a GENI project that it belongs to, and attributes of the project and slice from these authorities or other trusted parties. (For the GENI attributes we use SAFE logical certificates from synthetic authorities rather than standard GENI formats, but their content matches the GENI trust model [19].) The virtual SDX slice controller runs an instance of the SAFE logical inference engine to validate each request against configured security policies before approval.

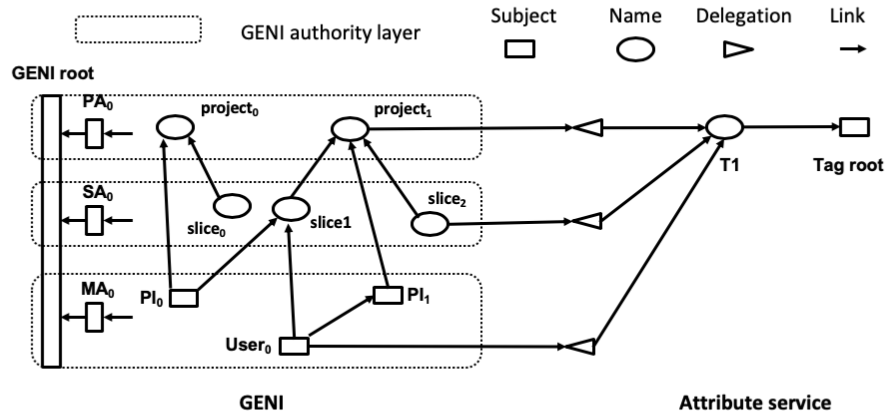



FIGURE 4.5: **Linking patterns for SAFE sets/certificates in the GENI-derived trust model with attribute service. Users, projects, and slices link to their endorsing authorities; slices link to their projects; users and leaders (PIs in the figure) link to the projects they own and/or have membership in; slices, projects and users link to the granted attributes.**

**Stitching.** A stitch request from another slice passes the *sliceID* of the requester. The *SliceID* can serve as a token for a certificate chain that identifies the slice, including any attributes and a binding to a project group, e.g., following the SAFE instantiation of the GENI trust structure [23, 19]. A hierarchy of testbed federation authorities govern the slices and projects, and assign security attributes to them. The ExoGENI provider API requires the "hard-to-guess" secret as a one-time passcode to validate mutual consent for cross-slice stitch requests [90]. A customer network (e.g., a campus) can also request a stitch at a named static stitchport with a named VLAN (network segment), if the NSP exposes static stitchports.

As customer slice requests for stitching to a virtual SDX provider's network, the virtual

**Table 4.1: Exemplary policies for network authorization in ExoPlex and their complexity and authorization throughput. The chain length is the inference depth of a typical logical proof of compliance for each policy.**

| Policy | Type | Description | Chain length | Throughput (K ops/s) |
|---|---|---|---|---|
| `approveByUserACL` | Stitch | Accept if the customer is on a virtual SDX access control list (ACL). | 5 | 2.05 |
| `approveByUserAttr` | Stitch | Accept if the customer is endorsed as eligible by a trusted party. | 10 | 1.21 |
| `approveBySliceAttr` | Stitch | Accept if the customer slice is endorsed as eligible by a trusted party. | 12 | 1.65 |
| `approveByProjectID` | Stitch | Accept if the customer slice belongs to an eligible GENI project (by ACL). | 6 | 1.96 |
| `approveByProjectAttr` | Stitch | Accept if the customer slice is endorsed as eligible by a trusted party. | 13 | 1.05 |
| `connectByUserAttr` | Transit | Connect two subnets if the owners have the require attributes. | 11 | 0.59 |
| `connectByProjectID` | Transit | Connect two subnets if the owners are members of an eligible project. | 11 | 0.88 |
| `connectByProjectAttr` | Transit | Connect two subnets if the owners are members of eligible projects endorsed by a trusted party. | 21 | 0.38 |

SDX applies its own policies to validate each request before installing each peering link. Listing 4.9 shows an exemplary logical policy that approves a stitching request if the user is on the ACL and the slice authority of GENI endorses the user to have control privilege over the user slice.

**Listing 4.9: An exemplary logical policy for stitching authorization by user ACLs.**

```
approveStitchByUserACL(?User,?Slice):-
  ?SA :=rootPrincipal(?Slice),
  ?SA: controlPrivilege(?User,?Slice,stitch,_),
  userAclEntry(?User).
```

**Connectivity.** Connectivity is off by default, and all flows are permissioned by policies of the endpoints. Transit across an SDX is enabled only for flows that comply with applicable customer policies. The customer network is responsible to route outbound packets of a permissioned flow into the L2 link to the SDX. The SDX ensures that any inbound packets

it routes onto the link are from permissioned flows. Listing 4.10 shows an exemplary logical policy for virtual SDX to authorize the connectivity between two subnets. The connectivity between two subnets is only allowed when both subnet owners permit it.

**Listing 4.10: An exemplary logical policy for authorizing connectivity between two subnets.**

```
connectByUserAttr(?Alice, ?Bob, ?IPa, ?IPb):-
  ownPrefix(?Alice, ?IPa),
  ownPrefix(?Bob, ?IPb),
  ?Alice: allowConnectionByUserAttr(?Bob),
  ?Bob: allowConnectionByUserAttr(?Alice).
```

The stitching and connectivity policies may include arbitrary attribute checks on the requesting client, slice, and/or project, and on the set of authorities trusted to assert these attributes. Table 4.1 lists some exemplary policies for virtual SDX stitching and customer connectivity.

If transit is approved based on connectivity policies, the virtual SDX controller finds a path for the connection and installs the routes via SDN. As a result, the established connectivity among customers is end-to-end and bidirectional, and limited to the authorized subnets.

## 4.5 Evaluating Logical Trust for Virtual SDX

**Authorization Performance.** We evaluate logical virtual SDX by running representative workloads on a cluster of SAFE instances loaded with the trust scripts for stitching and connectivity authorizations. Each SAFE instance is a Scala process serving a REST API to invoke its trust scripts. For these experiments, we evaluated the cost of logical trust with a multi-threaded load generator process that invokes the trust scripts directly according to synthetic request mixes designed to demonstrate and stress specific functions and behaviors in the virtual SDX scenario. The SAFE engine and scripts handle all certificate generation, validation, and logical policy compliance checking needed to implement these functions. The point is to show that these trust functions for a virtual SDX can be implemented compactly using scripted logical trust (about 330 lines after importing the GENI trust script), and that the resulting implementation is fast enough to use in practice.

We measure the throughput that each engine can achieve under heavy loads of logic queries that is representative of for stitching and connectivity authorizations. In this experiment, all certificates are cached before the queries. This gives us a view of inference costs.

In a real deployment these costs are spread across many servers (e.g., one per principal) in parallel: capacity scales with the size of the federation. The system's only fundamental scaling bottleneck is the underlying certificate store—a scalable key-value store. However, our bundling approach results in higher ratios in the logic set cache than the cloud servers would see in practice, reducing costs for fetches and signature checking.

We run SAFE engine instance runs on a four-core KVM (Intel Xeon CPU E5-260 v2 @ 2.20GHz) with 12 GB of RAM. We created synthetic governance principals like GENI root, trust root for IP allocation (RPKI root) and trust roots for attribute delegation (attribute root). The GENI root endorse other principals to create projects and slices following the GENI trust structure. The RPKI root allocates IP prefixes to users directly. And attributes are delegated to users and objects in two hops. In the experiment, we created 30 projects, 1000 slices, 10 attributes. The lengths of ACL are about 5 to 10, except for ACLs based on user ID.

We evaluate the throughput of different queries are shown in Table 4.1. The throughput numbers are related but only related with the chain length. As the inference cost also includes cost for indexing statements and matching against multiple entries in the ACL. Queries for transit/connectivity are typically more expansive as they authorize the request on behalf of both subnet owners. Since the authorization costs are in control plane, the results indicate that logical trust is fast enough to be implemented practically for virtual SDX or similar interdomain networking applications.

**Authorized Stitching and Connectivity.** As an demonstration experiment, we run virtual SDX with SAFE authorizations for stitching and connectivity on ExoGENI. The network topology is the same as in Figure 3.11. The virtual SDX service slice has two OVS nodes on two sites ("UFL" and "UNF") with a provisioned link connecting them. We

```
ubuntu@subnetA:~$ ping -c 2 192.168.30.2
PING 192.168.30.2 (192.168.30.2) 56(84) bytes of data.
64 bytes from 192.168.30.2: icmp_seq=1 ttl=62 time=226 ms
64 bytes from 192.168.30.2: icmp_seq=2 ttl=62 time=3.63 ms

ubuntu@subnetB:~$ ping -c 2 192.168.40.2
PING 192.168.40.2 (192.168.40.2) 56(84) bytes of data.
64 bytes from 192.168.40.2: icmp_seq=1 ttl=62 time=226 ms
64 bytes from 192.168.40.2: icmp_seq=2 ttl=62 time=3.77 ms

ubuntu@subnetC:~$ ping -c 1 192.168.20.2
PING 192.168.20.2 (192.168.20.2) 56(84) bytes of data.
From 192.168.30.1 icmp_seq=1 Destination Host Unreachable

ubuntu@subnetD:~$ ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
From 192.168.40.1 icmp_seq=1 Destination Host Unreachable
```

FIGURE 4.6: **Ping results between subnets. Only connections between compatible subnets are allowed.**

run virtual SDX controller, SAFE server and SDN controller on "UFL" rack. The latency between the two sites (network latency between the OVS at "UNF" and the SDN controller) is about 1.7 ms and the latency within the same site is negligible.

The virtual SDX authorizes stitching request by user ACL and customer subnets specify their connectivity policies that allow connections only to peer subnets with compatible security tags. In this experiment, virtual SDX allows stitching request from all four customers and the attribute root delegates attribute "tag0" to A and C, and delegates attribute "tag1" to B and D.

It takes about 9 minutes to stitch the four customer slices to the virtual SDX slice. The customer subnets advertise its route and connectivity policies after the stitching completes. Then I try to ping between subnets. As shown in Figure 4.6, only the connection between subnet A and subnet C and the connection between subnet B and subnet D are allowed.

The round trip of the first packet of a new connection takes longer time, because the SDN controllers has to report the event to the virtual SDX controller, and the virtual SDX controller has to authorize the connection and configure the dataplane accordingly. It takes

about 144 ms for the virtual SDX controller to authorize the connection request and 44ms to configure the routing path.

## 4.6 Summary

This chapter introduces the trust logic and SAFE trust management system, shows how to manage multidomain network security with trust logic and evaluate the performance of logical authorization for exemplary applications. As an example, we discuss security challenges for virtual SDX and demonstrate how ExoPlex manages network security for virtual SDX.

# 5

# Logical Peering for Interdomain Networking on Testbeds

NSPs on network testbeds can peer with one anther and provide end-to-end network service jointly, leading to interdomain networking similar to the Internet today, where NSPs peer with others for global connectivity. The subnet owner or its NSP needs to originate a route for the prefix. NSPs exchange routing information with their neighbors and learn available paths to the subnets, which is done via Border Gateway Protocol (BGP) in the Internet. Routing security is crucial for internetworks as malicious or misconfigured NSPs may attract traffic for dropping, tampering or spying with illegal route advertisement.

The logical trust framework in ExoPlex supports flexible network policies that are deployable with virtual NSPs on testbeds. It allows us to implement origin authentication (RPKI [21]) and route validation (BGPsec [80], SBGP [54]) with logic to protect the interdomain routing on testbeds. We can also explore other innovative and customized approaches for interdomain routing and evaluate them with virtual NSPs on network testbeds. For example, a community of science networks may specify customized policies that allow only approved NSPs to carry their traffic (see §5.3.2).

In this chapter, we will introduce secure policy-based inter-domain routing among *transit*

*NSPs* that extends Chapter 4 towards the goal for experimental NSP services in testbeds to carry real user traffic across research testbeds. In addition to existing approaches like origin authentication and route validation, it allows customer networks to specify policies about which NSPs can carry their traffic. We propose and demonstrate policy-based interdomain NSP networking in the ExoPlex toolkit—software elements that run within testbed slices and their controllers—to build NSPs and interconnect them securely. The discussion about the security and threat model is in §5.1. Then we demonstrate the design overview (§5.2), including secure routing and customized path control policies, and how ExoPlex composes and enforces path control policies. Then we discuss implementation details for SAFE routing with customized path control policies (§5.3). Last, we demonstrate experiments in §5.4. This chapter is adapted from [92] with permission.

## 5.1 Logical Peering in ExoPlex

**Security model for peering.** NSP controllers expose APIs to negotiate link stitching. An NSP's policies may limit the customers or peers that it accepts. Once a peering link is established, either side may advertise routes for subnet prefixes to the other. Secure inter-domain routing requires that NSPs validate prefix ownership (origin authentication) and transitive route advertisements end-to-end (route validation), similarly to Internet security standards such as RPKI [21] and S-BGP [53] or BGPsec [80]. We add customer-specified policies for off-by-default connectivity and *path control*, which limit traffic and constrain eligible routes based on security attributes of the NSPs and subnets.

The logic approach allows any participant to check compliance with another's policy on its behalf. For example, customers trust their edge providers (SDX) to enforce their connectivity policies. NSPs along a valid path cooperate to enforce customer-specified path control policies; the customer trusts these NSPs to be faithful to the policy.

As an exemplary demonstration, we deploy an inter-domain network with ten ExoGENI slices representing SDX, transit NSPs, and customer domains (Figure 5.1). In the demo scenario, customers specify path control policies that confine their traffic to compliant paths

through qualified carriers—NSPs endorsed with specified tags anchored in trust roots that the customer accepts. For example, an endorsing authority might issue a signed assertion tagging the NSP with public key $K$ as "production-grade safe" or "classified secure".

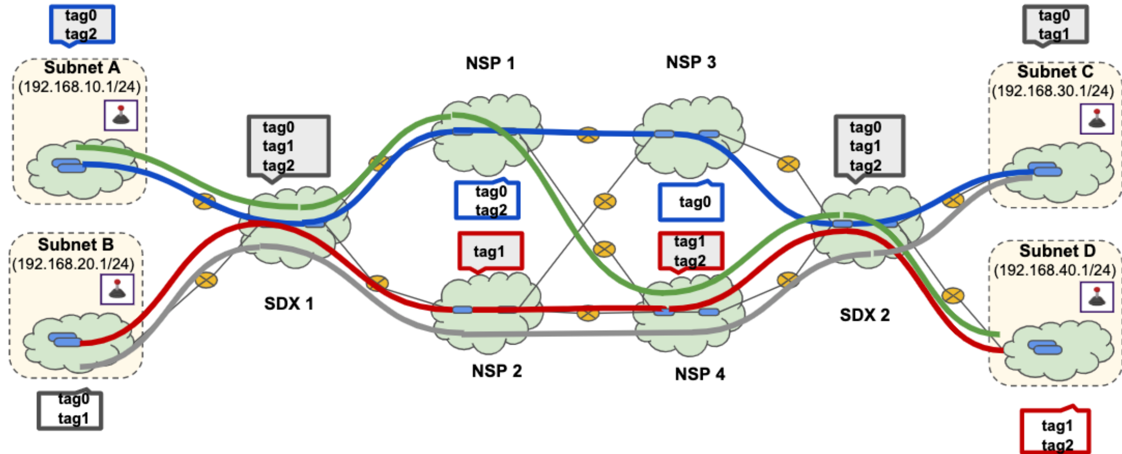



FIGURE 5.1: **An exemplary inter-domain network used in our experiments. Each of the participating NSPs is instantiated as a separate ExoGENI slice with an NSP controller and a principal keypair. The control plane comprises NSP controller APIs to establish peering links and to announce or propagate routes and security policies. Each customer edge subnet (stub) stitches to an edge provider (SDX) and specifies policies to control connectivity with other subnets and rules for packet transit, including** *path control* **policies that limit their traffic to qualifying NSPs endorsed with specified tags. For example, traffic transit between subnet A and subnet C is limited to only NSPs compatible with tag0. Similarly, B-D traffic is restricted by tag1; A-D traffic is restricted by tag2. As a result, certain flows take different paths in order to comply with each customer's traffic policies.**

**Standards and interoperability.** Networks base routing and security functions on well-specified protocol standards that allow for multiple interoperable implementations. In this work we take a first step by defining a common software platform—ExoPlex—that NSP controller software may use to manage their interactions and program their internal dataplane networks accordingly. Logical peering in the control plane offers alternatives to relevant Internet standards (e.g, BGPsec and RPKI), but with a simpler deployment for SDN-enabled testbeds, no dataplane entanglements, flexible governance, and extended policy options (e.g., path control as in our experiments). Because security metadata propagates through the control plane APIs over the public Internet, all crypto operations occur off of

the dataplane.

**Threat model.** We use logical peering to express rules that defend against IP spoofing, route hijacking, and unauthorized traffic. We provide standard logic rules for origin authentication and transitive route validation, modeling RPKI and BGPsec. We use path control to illustrate the potential for custom policies for logical peering. With path control, the transit path for each flow is compliant end-to-end with rules specified by the endpoints: the endpoints trust each NSP along the path to be faithful to the policies and to forward and accept traffic only along the trusted path, providing deep defenses against spoofing.

## 5.2 Design Overview

### *5.2.1 Secure Routing*

ExoPlex includes off-the-shelf trust logic scripts for secure routing, including certified route advertisements modeled on BGPsec and prefix ownership modeled on RPKI.

### *5.2.2 Path Control*

**Path control.** We add support for customers (subnet owners) to express logical policy rules for *path control*. These rules qualify which NSPs are eligible to carry their traffic, e.g., based on secure attributes of the NSPs. Interdomain routing in ExoPlex finds the least-cost paths that are compliant with registered policies of both the source and destination subnets, if such paths exist. A path (route) is compliant with the policy iff it traverses only qualified NSPs.

The subnet owner issues a path control policy as a certificate, and notifies its provider, passing the policy link. A policy notification associates each policy with a prefix pair `(source, dest)`, which may be wildcarded. The route for a packet is governed by the policy with the *most specific* enclosing prefix pair, if any, for the packet's `source, dest` addresses. If both source and destination assert a policy, then a compliant route complies with both.

**Inbound path control.** An *inbound* policy qualifies NSPs to carry traffic to a des-

tination prefix, and originates from the owner of the prefix. The subnet owner trusts the qualified NSPs, for example, to block any traffic to the destination from spoofed source addresses. Inbound path control policies are attached to a route advertisement and propagate with the route advertisement. Each NSP propagates routes only to peer NSPs that are compliant with the route's policy.

**Outbound path control.** An *outbound* policy qualifies NSPs to carry traffic from the source prefix $S$ (whose owner specifies the policy) to the destination prefix $D$. A customer passes outbound path control policies to the provider in a separate API call, which it may invoke at any time.

Upon receiving an outbound policy event, an NSP $N$ validates its default route for $(S, D)$ (if any) for compliance with the new policy. If the route is not compliant, then $N$ must find an alternative compliant route, even if it is longer than the current route, and then propagate it.

To do this, $N$ considers other cached routes to $D$. Consider a cached route $R$. $N$ received $R$ previously from a peer, but $N$ did not select or propagate $R$ because $N$ instead selected a shorter route (e.g., the current route). If $R$ is the shortest known compliant route, then $N$ selects $R$ for $(S, D)$, replacing the current route for any flows that are within the scope of the new policy. If $N$ knows no compliant route $R$, then it propagates the policy to at least all compliant peers that have advertised a valid route to $D$, indicating that the peer is also compliant with $D$'s inbound policy. These peers handle the event similarly.

Eventually, if a compliant path exists, some compliant NSP identifies a compliant $R$ and advertises it as described above. The route propagates in the usual fashion and eventually the SDX for $S$ receives it. Along the way, each NSP on the path chooses and installs a compliant sub-route $R$ for matching flows. If an NSP later learns of a shorter compliant route it replaces the old route in the usual way.

**Policy conflicts.** A route must comply with both the outbound policy of the source and the inbound policy of the destination. Conflicts are not a concern, although restrictive policies might block traffic entirely. If one subnet owner publishes conflicting policies for

different prefix pairs, then the longest prefix match takes priority. By convention, the source prefix dominates for an outbound policy and the destination prefix dominates for an inbound policy.

**Proof sketch for liveness.** The following conditions assure discovery of a compliant path, if one exists. First, each NSP that receives the outbound policy and has no compliant $R$ propagates the policy to all compliant peers. Second, all NSPs advertise each locally selected route to all compliant peers. Lastly, an NSP that has received the policy and knows a compliant route selects the route, and therefore propagates it to all compliant peers. Since a compliant path can never traverse a non-compliant peer, it is sufficient to propagate the policies and routes only among compliant NSPs. The liveness property can be proven by contradiction.

### *5.2.3 Policy Composition and Priority*

ExoPlex routes and policies are specified to match prefixes or source-destination $(S, D)$ prefix pairs; arbitrary ranges are not supported. Because it enables policies based on source address, it requires source-specific routing in the dataplane, and the NSP controller tracks policies and routes by $(S, D)$ prefix pairs, with optional wildcarding in either dimension. For any given set of routes and policies, the NSP must select a minimal set of source-specific routes to install in its SDN dataplane. Each installed route matches an $(S, D)$ prefix pair, and complies with policies applicable to $S$ and $D$. Thus a route may correspond to a region of overlap among multiple controlling policies. This section summarizes policy scope, overlap, and priority.

Figure 5.2 shows how two prefix pairs can overlap. A prefix pair $(S, D)$ corresponds to a rectangle (a *region*) in the 2-dimensional source-destination IP address space. In each dimension of two $(S, D)$ prefix pairs, the prefix of one prefix pair is either a subset or a superset of the corresponding prefix in the other pair. Thus the regions either cover or intersect as shown in Figure 5.2.

**Containment and priority** For simplicity, we require that prefix holders specify clear

priority for policies that conflict. For this reason, the only permitted form of overlap for policies of the same type (inbound or outbound) is containment—or equality. For example, Figure 5.3 (discussed below) presents a scenario in which an endpoint subnet specifies an outbound policy, overriding a policy for the containing network. Additionally, an endpoint's inbound policies must match its advertised routes: an inbound policy's destination prefix must be the same or smaller (more specific) than the prefix for some advertised route. These restrictions do not constrain the policy, only its specification.

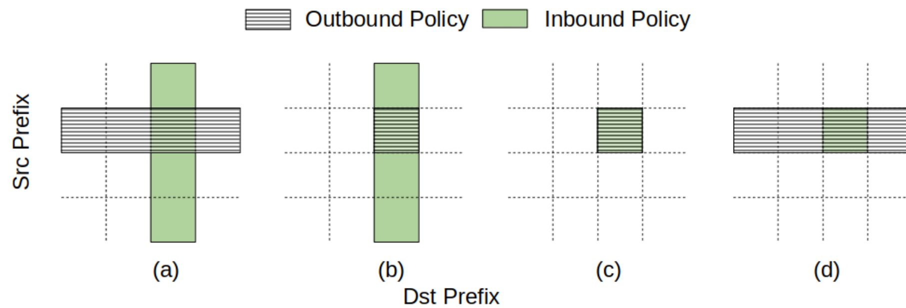



FIGURE 5.2: **Cases for two overlapping prefix pairs: they intersect, one pair contains the other, or they are equal.**

These containment properties simplify policy handling. Containment offers a clear priority rule: the more specific policy dominates. If an inbound or outbound policy is the highest priority of its type for some region, we say that the policy *controls* the region. Each point has exactly one controlling policy of each type; if no policy is specified, then the default is to accept any valid route. An NSP controller installs SDN (OpenFlow) routing rules matching each policy region, specifying the rule's priority as the area of the matching region. In this way the most specific policy applicable to a given packet determines its route.

Policy priority does not limit the flexibility of the policies. A network owner might limit its subnets to comply with parent policies, so that more specific policies are more restrictive. However, ExoPlex does not enforce such constraints. We leave it to network authorities to enforce compliance by their delegates at their discretion.

**Composing inbound and outbound policies** Policy composition occurs when an inbound policy and outbound policy match overlapping regions. Suppose a region $R$ is an area of overlap between an inbound policy and an outbound policy that both control $R$. Then compliance may require the NSP to select and propagate a different route for traffic matching $R$ than it selects for the other regions that these policies control. Specifically, a route for packets matching $R$ must comply with both the inbound and outbound policies that control the region. We refer to a pair of inbound and outbound policies that control the same region as a *policy pair*. Each point is governed by exactly one policy pair.

### *5.2.4 Processing Advertised Routes and Policies*

*Extended content.* This section outlines data structures and algorithms to manage policies indexed by $(S, D)$ region in the NSP controller. Each NSP maintains a catalog containing all of the routes and policies that it knows, indexed by region in an AQT. When it receives a new route or policy, it queries its catalog to determine adjustments to its current routes, and how to propagate the policy and affected routes to its peers. For simplicity, we discuss propagation of policies and routes separately.

**Table 5.1: NSP controller data structures used by Algorithm 1 and Algorithm 2.**

| | |
|---|---|
| $inbound$ | AQT store for inbound policies indexed by region. |
| $outbound$ | AQT store for outbound policies indexed by region. |
| $match$ | AQT store for overlapping regions of controlling policy pairs. |
| $routes$ | Store of all accepted routes from neighbors. |
| $forwardMap$ | Map of current chosen routes and their regions. It is similar to forwarding information base (FIB), but at NSP level. |
| $exports$ | Set of routes exported to compliant neighbors. |

**Route and policy matching**. Area-based Quad Tree (AQT) [20] supports efficient indexing of regions specified by prefix pairs. The root of the AQT represents the entire 2-D address space. Each AQT node has four children that equally split the parent's address

space, adding one address bit in each dimension. Thus the nodes of the AQT correspond to progressively finer-grained squares in the 2-D space. In [20] a region or prefix pair stored in the AQT is also called a *filter*. The AQT stores each filter in the highest-level node (closest to the root) that shares the same source or destination prefix, whichever dimension of the region is larger (less specific). As [20] explains, each region is a *crossing filter* for the node it occupies: the region is either identical to the node's square in the 2-D space, or the region exactly crosses the square in exactly one dimension. The filters stored at a node are the node's *crossing filter set*.

Each populated node of the AQT has two collections to index its crossing filter set, one for each dimension. It stores crossing filters in the collection that indexes the smaller (less specific) dimension of the filter. We choose a binary tree to represent each collection, where each node in the binary tree represents a prefix. To insert or remove a filter, the AQT walks from the root to the target node and updates the node. To query a prefix pair, the AQT returns all stored filters that overlap with the pair's region, leaving any conflict resolution to the routing management module.

For $N$ prefix pairs, AQT requires $O(N)$ space, $O(W)$ update (insert/delete) time, and $O(N)$ query time, where $W$ is the maximum prefix length—32 for IPv4 prefixes.

**Algorithm overview: new policy**. At any given time there is a set $P$ of filters of a first type (inbound or outbound), a set $R$ of filters of the other type, and a set $M$ of *match* regions. Each set $P, R, M$ is indexed in its own AQT.

Different policy pairs may match on the same region, but each match is controlled by exactly one most specific policy pair: $\forall m \in M,\ \exists p_m \in P,\ \exists r_m \in R,\ m = p_m \cap r_m \wedge \forall p \in \{p \in P \mid m \subseteq p\}, p_m \subseteq p \wedge \forall r \in \{r \in R \mid m \subseteq r\}, r_m \subseteq r$. *Proof.* Let $X$ be the set of all points in 2-D address space. Because of the containment property, there is a most specific filter $p_x$ in $P$ and a most specific filter $r_x$ in $R$ that control each point $x$. The most specific filter may be the default filter. The most specific policy pair that controls $x$ is then $(p_x, r_x)$. The corresponding match region $m_x = p_x \cap r_x$ is the minimal match region that controls $x$: $\forall x \in X,\ \wedge \forall m \in \{m \in M \mid x \in m\},\ m_x \subseteq m$.

Consider addition of a new filter $p$ of the first type. Existing matches $m \in M$ that do not overlap $p$ ($m \cap p$ is empty) are unaffected. If $p \in P$ then for each $r \in R$ that overlaps $p$ the algorithm must visit the existing match $m = p \cap r$ to determine if it should use the new policy. If $p \notin P$ then each such $r \in R$ may create a new match region $m = p \cap r$ to add to $M$.

Whether or not $p \in P$, there may exist one or more matches $m$ that overlap: $m = p_i \cap r_j$ with $p_i \in P, r_j \in R, p_i \neq p$, $m \in M$, and non-empty $p \cap m$. Because of the containment property, for any such $m$, there are exactly two cases. Case 1: $p_i \subset p$. Then $p_i$ dominates $p$, so the new $p$ does not affect this $m$. Case 2: $p \subset p_i$. Then the new $p$ introduces a match filter $m' = p \cap r_j \subseteq m$ that dominates $m$ for sub-region $m'$. Filter $m$ is unaffected and remains in place to control the rest of its region. It is possible that $m' = m$ if $r_j$ itself is more specific than $p_i$ in at least one dimension, but $p$ updates policy for $m'$ regardless.

The algorithm handles all of these cases by considering $m = p \cap r$ for each overlapping $r \in R$ in priority order where such order exists. If $m \notin M$, then add $m$ to $M$ controlled by policy pair $(p, r)$. Else $m \in M$ and $m = p_i \cap r_j$ for unique $p_i$ and $r_j$ as described above. If $p_i \subset p$ then there is no change for this $m$ because $p_i$ dominates the new $p$ in this region. If $p_i = p$ or $p \subset p_i$, then the new policy supersedes the old one in $m$. However, if $r_j \subset r$ then there is no need to update $m$: $r_j$ has higher priority than $r$, so the algorithm has already considered $p \cap r_j$ and so has already added or updated $m$ for $p$.

**New outbound policy**. Algorithm 1 shows the NSP procedure to process a received outbound policy $policy_{ob}$ with the specified region and policy certificate. The NSP stores the outbound policy in *outbound* AQT indexed by its region. Then it runs a query to the *inbound* AQT to retrieve a list of all applicable inbound policies that overlap with $policy_{ob}$ (line 3), in descending order of policy priority (ascending order of region area). Then for each inbound policy $policy_{ib}$ in the list, it computes the overlapped region of the inbound policy and the new outbound policy. It then queries the *match* AQT for this region (line 6).

If the region is not present in *match*, then add it for this policy pair: the matched

inbound policy $policy_{ib}$ and the new policy $policy_{ob}$ control the region. Suppose instead that there exists a policy pair in *match* with the same overlapped region. If $policy_{ib}$ is the same inbound policy as in the pair that is most specific for the region, and $policy_{ob}$ has equal or higher priority (more specific) than the outbound policy of that pair, then the new policy $policy_{ob}$ controls: update the policy pair for the region (lines 7-8). Otherwise, at least one of $policy_{ob}$ or $policy_{ib}$ is of lower priority (less specific) than the corresponding policy of the pair. If it is $policy_{ob}$, then we do not update the region because the previous outbound policy dominates in this region. If it is $policy_{ib}$ but not $policy_{ob}$, then we have already processed a more specific inbound policy dominating the overlapped region.

If the new $policy_{ob}$ controls any region, then find a (new) compliant route for the region. The NSP retrieves all accepted routes whose destination prefixes fully cover the destination prefix of the overlapped region in order of preference; currently the preference order is by path length, but the NSPs are free to prefer routes by other attributes. If a route is the most preferred among compliant routes to the policy pair of the region, the NSP chooses the route for traffic in the overlapped region and exports the route if it has not done so already (line 11-19). If there is no compliant route for the pair, the NSP drops any matching traffic in the overlapped region.

Figure 5.3 shows an example scenario in which NSP 5 processes a new outbound policies from NSP 6, which requires a different path for the specified region. Suppose that NSP 5 have learned all routes and policies except the outbound policy from 6 at the beginning. Then 6 advertises its outbound policy $o_1$ and NSP 5 chooses a different path for the region accordingly. Table 5.2 shows the states of NSP 5 at different stages.

**Table 5.2: The states of NSP 5 before and after processing the outbound policy from 6 in Figure 5.3.**

| event | routes | inbound | outbound | match | forwardMap | exports |
|---|---|---|---|---|---|---|
| start | $r_0$ : 1.1.1.0/24, [2, 1]<br>$r_1$ :1.1.1.0/24, [4, 3, 1] | $i_0 : \langle *, 1.1.1.0/24, any \rangle$ | $o_0 : \langle *, *, any \rangle$ | $\langle *, 1.1.1.0/24 \rangle \rightarrow \langle i_0, o_0 \rangle$ | $\langle *, 1.1.1.0/24 \rangle \rightarrow r_0$ | $r_0$ |
| $o_1$ | $r_0$ : 1.1.1.0/24, [4, 3, 1]<br>$r_1$ : 1.1.1.0/24, [2, 1] | $i_0 : \langle *, 1.1.1.0/24, any \rangle$ | $o_0 : \langle *, *, any \rangle$<br>$o_1 : \langle 2.2.0.0/16, *, \text{“}s\text{”} \rangle$ | $\langle *, 1.1.1.0/24 \rangle \rightarrow \langle i_0, o_0 \rangle$<br>$\langle 2.2.0.0/16, 1.1.1.0/24 \rangle \rightarrow \langle i_0, o_1 \rangle$ | $\langle *, 1.1.1.0/24 \rangle \rightarrow r_0$<br>$\langle 2.2.0.0/16, 1.1.1.0/24 \rangle \rightarrow r_1$ | $r_0$<br>$r_1$ |

**Algorithm 1** Process a new outbound policy. See Table 5.1 for notation. The *query* method of the AQT returns a list of objects whose regions overlap with the queried region.

```
1:  event policy_ob(region, cert) :
2:  outbound.put(policy_ob.region, policy_ob)
3:  ibPolicies = inbound.query(policy_ob.region)
4:  for policy_ib ← ordered(ibPolicies) do
5:      region_ol = policy_ob.region ∩ policy_ib.region
6:      ⟨policy_ob^prev, policy_ib^prev⟩ = match.get(region_ol)
7:      if region_ol ∉ match or (prio(policy_ob) ≥ prio(policy_ob^prev) and
    policy_ib == policy_ib^prev) then
8:          match.put(region_ol, ⟨policy_ob, policy_ib⟩)
9:          orderedRoutes = routes.filter(r → region_ol.dst ⊆ r.dst)
10:         bestRoute = null
11:         for route ← orderedRoutes do
12:             if compliant(route, policy_ib, policy_ob) then
13:                 bestRoute = route
14:                 if route ∉ exports then
15:                     exports.add(route)
16:                 end if
17:                 break
18:             end if
19:         end for
20:         forwardMap.put(region_ol, bestRoute)
21:     end if
22: end for
```

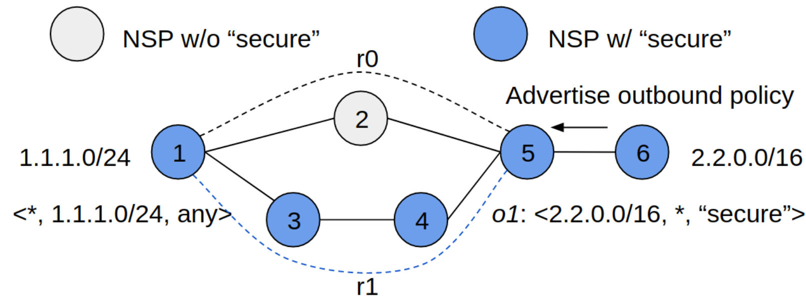



FIGURE 5.3: **An example scenario in which policy overlap forces selection of an alternate route for the intersection. NSP 5 processes a new outbound policy $o_1$ from 6 that requires NSPs with attribute “secure” for its outbound traffic, forcing matching traffic to the longer path (in blue).**

**New inbound policy**. The procedure to process a new inbound policy is similar to Algorithm 1.

**New route**. Algorithm 2 shows the procedure to process a new route that the NSP accepts. First, the NSP retrieves all policy pairs for overlapped regions by the destination prefix. We use the route only if the destination prefix of the overlapped region is covered by the advertised destination prefix in the route. If the new route is compliant to the policy pair and is preferred over the existing best route (which may be empty), the NSP chooses the new route for traffic matching the overlapped region and exports the route to its neighbors.

**Algorithm 2 Process a new accepted route.**

1: **event** $route(dst, path)$ :
2: $routes.add(route)$
3: $policyPairs = match.query(\langle *, dst \rangle)$
4: **for** $\langle policy_{ob}, policy_{ib} \rangle \leftarrow policyPairs$ **do**
5: $\quad region_{ol} = policy_{ob}.region \cap policy_{ib}.region$
6: $\quad$ **if** $region_{ol}.dst \subseteq route.dst$ **then**
7: $\quad\quad route^{prev} = forwardMap.get(region_{ol})$
8: $\quad\quad$ **if** $compliant(route, policy_{ib}, policy_{ob})$ **and** $pref(route) > pref(route^{prev})$ **then**
9: $\quad\quad\quad forwardMap.put(region_{ol}, route)$
10: $\quad\quad\quad exports.add(route)$
11: $\quad\quad$ **end if**
12: $\quad$ **end if**
13: **end for**

The forwarding path of the highest-priority (most-specific) OpenFlow entry for a packet in dataplane must comply with the most specific policy pair (match region) that controls its source and destination IP address. Algorithm 1 ensures this property because: (1) it considers every possible match region by processing every inbound policy that overlaps with the new outbound policy; (2) it always identifies the most specific match by ordering inbound policies in descending priority; (3) it updates the policy for a match region only if the new policy is more specific than the existing policy. Given that property, routing is correct because: (1) the controller installs an OpenFlow entry for each match region; (2) it

assigns the priority of the entry as the area of the corresponding region. Thus OpenFlow entries that match on more specific (smaller) regions have higher priority in the dataplane.

#### *5.2.5 Source-specific Routing with SDN-enabled Dataplane*

To support inbound and outbound path control policies, we need to implement source-specfic routing that match on both source and destination IP address in SDN-enabled dataplane. We can run OVS in ExoGENI slice to support SDN. Testbeds like Chameleon and ESnet also deployed hardware switches (Corsa DP2000 series) that support SDN and virtualization. They allow users to manage isolated networks with OpenFlow-enabled virtual forwarding contexts (VFC) and their own OpenFlow controllers.

Source-specific routing requires more matching fields in OpenFlow entries and potentially more OpenFlow entries. We evaluate the overhead of source-specific routing in OVS in §5.4.5. The matching fields and the number of OpenFlow entries in hardware flow table do not affect the performance of hardware switches [15], as they can match different OpenFlow entries in parallel with TCAM (ternary content-addressable memory). However, hardware switches come at higher economic cost and have limited hardware flow table capacities.

## 5.3 Implementation

This section presents an extended treatment of the ExoPlex prototype. We focus on how it uses logical trust to implement a trust plane for secure interdomain networking.

ExoPlex is designed to operate across inter-connected SDN networks (dataplanes), including virtual SDN networks hosted on testbeds. We make minimal assumptions about the capabilities of their switch infrastructure or virtual hosting services. Each ExoPlex NSP is under the control of a domain. The NSPs correspond to autonomous systems (AS) in Internet BGP routing protocols. We use a different term to avoid confusion with those protocols, since ExoPlex does not use BGP or assume any support for BGP. Instead, it implements secure routing and security policy using authenticated logic exchanges in the control plane, as described below.

Thus ExoPlex is compatible with a range of testbeds and SDN systems. Its control plane and trust plane are entirely decoupled from the protocols used to operate the SDN dataplanes: it operates above the SDN networks and is unknown to them. It is also transparent to the hosting providers (e.g., testbeds), because NSPs use the providers' native APIs to manage virtual infrastructure for the NSP dataplane. For example, on ExoGENI, an NSP may use APIs in the Ahab toolkit to provision its nodes and links [90], and APIs for cross-slice peering to stitch NSP dataplane networks at L2 via the *stitchport* abstraction or via direct peering at a common hosting site [86]. Customer networks may be campus subnets that connect to supported transport fabrics (e.g., I2-AL2S, ESnet) and can attach to stitchports.

### *5.3.1 Control Plane*

The NSP controllers export RPC APIs (e.g., REST/HTTP) to control inter-domain peering and networking. The customers and peers invoke these APIs summarized in table 3.1 to attach (stitch) an L2 link and to enable specified IP traffic to flow over the link. These calls propagate the routes and policies that govern traffic flow, which are encoded in logical certificates.

### *5.3.2 Example Scenario: FabNet*

We consider an example ExoPlex scenario: FabNet, a hypothetical secure internetwork for a community of scientists. It comprises an assemblage of network resources spanning multiple campuses and research fabrics. These resources are allocated and programmed for use by researchers in some field. FabNet traffic crosses multiple network providers (NSPs) resident on those fabrics, as well as the campus networks at the edges. A FabNet consortium approves and endorses participating campuses and NSPs.

Traffic traverses FabNet by agreement of the sender and receiver of the traffic. The endpoints are subnets on the attached campuses. Suppose a campus network authority (CNA) assigns an IP prefix to a secure subnet and delegates ownership and limited control

of the subnet to a research group. The research group leader issues a request to enable connectivity with a collaborator's subnet on a peer campus. The endpoints agree that traffic between them shall traverse FabNet.

The campuses and FabNet NSPs cooperate to direct selected IP traffic through FabNet. To attach to FabNet, a campus network operator establishes circuit connectivity to a selected FabNet NSP edge site. They cooperate to validate all prefixes and routes to ensure that the traffic is authorized for FabNet, and that it transits only approved FabNet NSPs.

**FabNet governance**. This example features multiple governance authorities and other identities interoperating with the NSPs. The governance structure defines a natural PKI hierarchy via logical trust. For example, the FabNet consortium root acts as a shared trust anchor whose PID is known to all participants. It endorses the PIDs of the CNAs and NSPs. The CNAs also endorse campus subnets, assert attributes of subnets, and delegate selected management authority to researchers that control those subnets. Ownership of the containing prefixes is certified by a delegation hierarchy rooted in FabNet or some other authority, e.g., ICANN. Section 5.3.6 discusses governance in more detail.

### *5.3.3 Secure Routing with Logical Trust*

The ExoPlex prototype includes SAFE trust scripts used by all participants in an ExoPlex network. The scripts embody sample logic for the secure routing mechanisms and policies in this chapter. Specifically, they include standard datalog logic rules to secure all routing with origin authentication (following RPKI) and route authentication (following BGPsec).

The prototype also includes exemplary policy logic rules and certificate templates that endpoints may use to authorize peering, connectivity, routing (path control), and governance. The exemplary policies are sufficiently powerful to implement FabNet and other protected networks over a set of NSPs. Crucially, these elements do not affect the NSP controllers or their API in Table 3.1. Instead, various principals invoke issuer scripts to issue linked certificates and policy rules to validate them. The endpoints may define new policy types by programming new assertion types and validation rules into their trust scripts. The

NSP controllers invoke standard guard scripts that import the certificate DAGs and query them for compliance under the applicable rules. Our approach can also accommodate NSP routing policies, e.g., to prefer certain routes or exclude unauthorized traffic, but we do not discuss NSP policies in this chapter.

The exemplary policy logic is based on Attribute-Based Access Control (ABAC). Authorities use a common logic package for ABAC to generate attributes, assign or delegate them to other principals, and check that specified attributes are present. The implementation represents attributes as *tags*. A tag is a string name that represents a permission, role, group, or attribute. A logic policy may state that a principal is authorized if and only it wields or possesses the tag, or perhaps a conjunction or disjunction of tags. For example, a subnet owner may express policies that restrict the set of NSPs that can carry its traffic, based on their tags.

Each tag has a controlling authority—a principal who creates the tag and defines the rules for delegating the tag. Any principal may create tags and act as a tag authority. The tag authority (or root) has sole power to specify which principals wield the tag. In the implementation, tags are self-certifying: the tag name is a concatenation of its root's PID and a name (such as a UUID) chosen by the tag root. An authorizer accepts a tag delegation as valid only if the tag's root asserts it or accepts it under its rules.

### *5.3.4 Authorization for the NSP API*

**Routing**. Traffic is also subject to origin authentication and route validation for secure routing, and endpoint path control policies to qualify all NSPs in the path. NSPs apply related guard checks on both sides of API calls to propagate routes and routing policies. The remaining subsections focus on these aspects.

### *5.3.5 Route Validation*

The NSP API in Table 3.1 allows peers and customers to attach to an NSP dataplane at specified peering points, register attached subnets, and enable traffic flows between prefix

pairs. After attaching, a neighbor may advertise a new prefix or routing policy at any time by passing a certificate link (token) through the API. The receiving NSP validates it, integrates it with its routing base, installs SDN rules to implement it in its data plane, and propagates it to peer NSPs through their APIs.

**Listing 5.1: The path control policy of customer network. The path is compliant to the routing policy of customer network iff each hop of the path is authorized by the customer policy.**

```
compliantPath(?SrcIP, ?DstIP, ?Path) :-
  eq([?Head|?Tail], ?Path),
  eq(?Tail, []),
  authorizedAS(?SrcIP, ?DstIP, ?Head).

compliantPath(?SrcIP, ?DstIP, ?Path) :-
  eq([?Head|?Tail], ?Path),
  authorizedAS(?SrcIP, ?DstIP, ?Head),
  compliantPath(?SrcIP, ?DstIP, ?Tail).

authorizedAS(?SrcIP, ?DstIP, ?Head):-
  nspTagAclEntry(?SrcIP, ?DstIP, ?Tag),
  tagAccess(?Tag, ?AS).
```

ExoPlex uses off-the-shelf rules (§4.2.2) for secure routing. And subnet owners can specify path control policies for its traffic and link them to the route advertisement. Listing 5.1 shows an example inbound path control policy that allows NSPs with specific tags to carry traffic from the source prefix to the subnet. Listing 5.2 shows a template script for a prefix owner to endorse NSPs with the specific tag to carry traffic from the source prefix to its prefix.

**Listing 5.2: A template script for customer network to specify the required attributes of NSPs for its inbound traffic.**

```
defcon inboundPolicy(?Tag, ?Src, ?Dst) :-
  {
    nspTagAclEntry($Tag, $Src, $Dst).
    label("nsp-tag-acl").
  }.
```

### *5.3.6 Discussion: Governance*

As described in §4.2.2, the off-the-shelf rules for secure routing in ExoPlex depend on additional logic for governance policy. A governance policy is a set of logical statements comprising facts to identify roots of authority and rules to validate certified delegations of authority from those roots. Each participant executes code to configure its own policy by

installing a set of logic statements. Of course, interaction depends on compatible policies: the security logic blocks unsafe interactions that conflict with a participant's policy.

Rather than specify a "one size fits all" governance structure, we use logical trust as a foundation to build and evolve governance structures and enable customers to specify their policy within those structures. Governance in the demonstration experiments works as follows. All principals are represented by a keypair. Slices and projects are objects approved by a controlling authority, which may make statements about them.

- A common federation root endorses a set of campus network authorities (CNA), one for each participating campus or enterprise. The root also publishes a common governance policy (see below).
- A common authority for the IP address space delegates disjoint prefixes to CNAs. The CNAs delegate sub-prefixes to owners of edge subnets.
- Subnet owners control traffic on their subnets, e.g., they may install the prefix on a testbed slice and/or "opt in" to an edge NSP (an SDX) $S$ by issuing network bypass commands to stitch to $S$ at L2, register $S$ as the local gateway for traffic to selected prefixes, and accept traffic sourced from selected prefixes from $S$.
- Once attached, subnet owners may advertise routes and publish path control policies to $S$.
- Testbed slices have security metadata mirroring the idealized federation governance structure for GENI [19]. NSP authorization policies for stitching may reference this metadata.
- A common network authority endorses NSP public keys, binds them to slices, and asserts security attributes of NSPs for use in path control (§5.2.2). Local policies may accept other endorsing authorities on a per-attribute basis.

Any participant may validate compliance with its locally accepted governance policy. Delegations result in chains or DAGs of linked logical certificates. The governance rules represent safety predicates for these structures. Any principal can apply its own rules to check validity for itself end-to-end, or delegate checking to another party, such as an edge

NSP that acts as its provider. Local governance policies are freely mobile: given a link to a policy of another principal, it is easy to import the policy's facts and rules and apply them.

This property of the logic system also makes it easy for a participant to delegate their governance policy to an authority. In our demonstration prototype, all participants subscribe to a common package of governance logic posted by a common root of authority that is trusted by all parties. The governance policy set installed by the local operator at each participant comprises simply a link to the remote policy set and a statement that accepts its conclusions: "If the policy authority's rules conclude $P(x)$ then believe $P(x)$."

## 5.4 Experiments

The ExoPlex prototype is suitable for experiments with secure routing and policy flexibility involving modest numbers of customer prefixes and NSPs. We conducted demonstration experiments on the ExoGENI testbed and I2-AL2S research fabric. The performance and scale of these experiments are bounded primarily by the current limitations of the fabric, the VM-based OpenVSwitch routers we use on ExoGENI, and the Ryu SDN controllers for each NSP. All compliance checks and crypto operations are off of the dataplane, and so affect only the setup times, and not the transit performance. There is an obvious tradeoff between policy granularity and scale: fine-grained policies lead to fragmentation of the prefix space and routing/SDN flow tables, which could be a scaling barrier.

We conducted several demonstration experiments based on the topology shown in Figure 5.1, involving ten ExoGENI slices. Each slice is instantiated under its own keypair and runs with its own controller, SDN, and SAFE logic engine, as in Figure 3.7. The customer networks A, B, C and D advertise subnet prefixes delegated to them through common governance authorities, which also endorse the networks with various security properties (tags), as described and shown in the figure. The NSP controllers interact via REST APIs to peer and propagate edge-to-edge routes and policies.

table=0, n_packets=1, ip, nw_dst=192.168.10.0/24
table=0, n_packets=1, ip, nw_dst=192.168.30.0/24

**(a) Flow table of $N1$.**

table=0, n_packets=1, ip, nw_dst=192.168.20.0/24
table=0, n_packets=1, ip, nw_dst=192.168.40.0/24

**(b) Flow table of $N2$.**

FIGURE 5.4: **Flow tables of NSP switches in experiment 1. We dump the NSP flow tables manually, filter the output flow entries by IP prefixes and packet counters for clarity, and omit unnecessary fields and the tables for $S1$ and $S2$. The packet counters confirm that traffic between subnets A and C traverses the path $\langle S1, N1, S2\rangle$ and traffic between subnets B and D takes the path $\langle S1, N2, S2\rangle$.**

### *5.4.1 Experiment 1: Inbound Path Control*

We evaluated inbound path control for the scenario shown in Figure 5.1, but omitting NSPs $N3$ and $N4$. On ExoGENI it takes about 6.5 minutes to provision this topology and stitch the peering links, limited by provisioning times for I2-AL2S circuits. The customer subnets stitch to their SDX providers concurrently, and then advertise their routes and path control policies when the stitches complete. NSPs validate those advertisements and policies, configure their dataplanes via SDN accordingly, and propagate them to their peers.

For this experiment, customers A and C both authorize only the NSPs with secure attribute “tag0” to carry their inbound traffic, and authorize connectivity only with edge subnets bearing the same attribute “tag0”. Customers B and D similarly authorize only “tag1” for their network traffic. Upon receiving the route advertisement with linked inbound policy from A, $S1$ propagates the route to $N1$ only, based on A’s policy. $N1$ validates the route, adds it to its cache of known routes, and propagates it to its authorized neighbor. Other routes and policies are advertised and propagated throughout the network similarly. Then, customer pairs A-C and B-D each request a flow to the partner.

It takes about 3 seconds for each route advertisement to propagate throughout the network and enable flows between the pairs. After these requests complete, we ping between subnet pairs with 1 packet and dump the flow tables from NSP switches, shown in Figure 5.4,

to verify that traffic follows the compliant paths as shown in Figure 5.1.

### *5.4.2 Experiment 2: Outbound Path Control*

For this experiment we extended the inter-domain network with NSPs $N3$ and $N4$, as shown in Figure 5.1, and added outbound policies. The routing policies in our experiment settings are shown in Table 5.3.

**Table 5.3: The inbound and outbound path control policies of subnets in Figure 5.1. In this setting, the inbound and outbound path control policies of a subnet are symmetric (but not necessarily symmetric), i.e., they require the same set of security tags for NSPs. The route between two subnets must be compliant to the path control policies of both the source subnet and destination subnet. For example, "tag 0" is the only legal security tag for routes between subnet A and C.**

| Subnet Owner | Inbound/Outbound Policies | |
|---|---|---|
| | Src/Dst Subnet | Required Tags |
| A | C | tag0 |
| | D | tag2 |
| B | D | tag0, tag1 |
| B | C (only at step 6) | tag0 |
| C | A | tag0, tag1 |
| D | A | tag2 |
| | B | tag1 |

We carry out the experiment in the following steps: (1) Stitch all NSPs. (2) Stitch subnets A, B and C to its NSPs. (3) Subnets A, B and C advertise routes with both source and destination address specified and their outbound policies. The NSPs authorize and propagates those routes and policies and make dataplane configurations accordingly. (4) Stitch subnet D to $S2$ and advertise its routes and policies. (5) Stitch $S2$ and $N2$ directly to provide a shorter routes for subnet B to reach subnet C and D. (6) Subnet B and C advertise inbound and outbound path control policies that require "tag0" for traffic between their subnets. It takes about 12 minutes to stitch all NSPs and subnets A, B and C. Figure 5.5 shows a timeline of the experiment starting at step (3).

We try sending a packet between connection pairs A and C, A and D, B and C, as well as B and D each time after step 3, 4, 5 and 6. Table 5.4 shows the paths that different

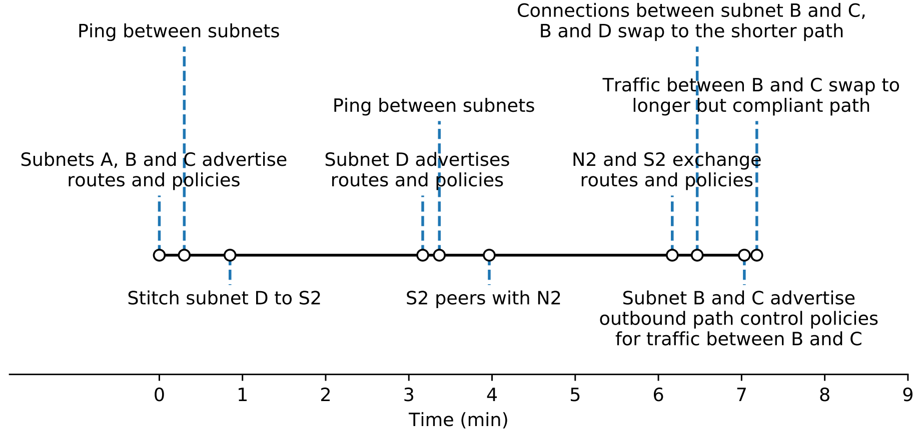



FIGURE 5.5: **Timeline of the experiment for outbound path control.**

flows take at different steps. The paths are compliant to the path control policies of the subnets as expected. Figure 5.6 shows the flow tables of NSP switches after step 6. The packet counters of the flow tables proves the correctness of Table 5.4. Before step 6, traffic between subnet B and C are subject to the default policies that require "tag0" or "tag1" for the NSPs. In our run, traffic between subnet B and C took the route $\langle S1, N2, N4, S2 \rangle$ before step 5 and $\langle S1, N2, S2 \rangle$ when the shorter route became available at step 5. After step 6, only the route $\langle S1, N1, N3, S2 \rangle$ is compliant for traffic between subnet B and C. We verify the actual route that those packets take by checking the packet counters of the flow tables of the NSPs after each step.

**Table 5.4: The path for traffic between subnet pairs at each step.**

<table>
<tr><th>Flow</th><th>3</th><th>4</th><th>5</th><th>6</th></tr>
<tr><td>A↔C</td><td colspan="4">$\langle S1, N1, N3, S2 \rangle$</td></tr>
<tr><td>A↔D</td><td></td><td colspan="3">$\langle S1, N1, N4, S2 \rangle$</td></tr>
<tr><td>B↔C</td><td colspan="2">$\langle S1, N2, N4, S2 \rangle$</td><td>$\langle S1, N2, S2 \rangle$</td><td>$\langle S1, N1, N3, S2 \rangle$</td></tr>
<tr><td>B↔D</td><td></td><td>$\langle S1, N2, N4, S2 \rangle$</td><td colspan="2">$\langle S1, N2, S2 \rangle$</td></tr>
</table>

### *5.4.3 SAFE Routing Authorization Performance*

**Throughput** We conducted experiments to evaluate the cost of logical authorizations in

```
n_packets=4,nw_src=192.168.10.0/24,nw_dst=192.168.30.0/24
n_packets=4,nw_src=192.168.30.0/24,nw_dst=192.168.10.0/24
n_packets=3,nw_src=192.168.40.0/24,nw_dst=192.168.10.0/24
n_packets=3,nw_src=192.168.10.0/24,nw_dst=192.168.40.0/24
n_packets=1,nw_src=192.168.20.0/24,nw_dst=192.168.30.0/24
n_packets=1,nw_src=192.168.30.0/24,nw_dst=192.168.20.0/24
```

**(a) Flow table of $N1$.**

```
n_packets=3,nw_src=192.168.40.0/24,nw_dst=192.168.20.0/24
n_packets=3,nw_src=192.168.20.0/24,nw_dst=192.168.40.0/24
n_packets=3,nw_dst=192.168.20.0/24
n_packets=3,nw_dst=192.168.30.0/24
```

**(b) Flow table of $N2$.**

```
n_packets=4,nw_src=192.168.10.0/24,nw_dst=192.168.30.0/24
n_packets=4,nw_src=192.168.30.0/24,nw_dst=192.168.10.0/24
n_packets=1,nw_src=192.168.30.0/24,nw_dst=192.168.20.0/24
n_packets=1,nw_src=192.168.20.0/24,nw_dst=192.168.30.0/24
```

**(c) Flow table of $N3$.**

```
n_packets=3,nw_src=192.168.40.0/24,nw_dst=192.168.10.0/24
n_packets=1,nw_src=192.168.40.0/24,nw_dst=192.168.20.0/24
n_packets=1,nw_src=192.168.20.0/24,nw_dst=192.168.40.0/24
n_packets=3,nw_src=192.168.10.0/24,nw_dst=192.168.40.0/24
n_packets=2,nw_dst=192.168.20.0/24
n_packets=2,nw_dst=192.168.30.0/24
```

**(d) Flow table of $N4$.**

FIGURE 5.6: **Flow tables of NSP switches in experiment 2 after step 6.**

isolation. We evaluated the inference performance of a SAFE server for validating routes and checking policy compliance under a throughput-limited synthetic workload. The figure of merit is authorization ops per second (authz-ops/sec) for the checks performed by an NSP when it receives an advertisement. We run the SAFE server on a machine with 16 2.6 GHz cores (Intel Xeon E5-2650 v2) and saturate it with concurrent authorization queries through its REST API. The evaluation measures the cost to process the network calls and the cost to run the logic query on logic content extracted from a linked certificate DAG.

We generated a synthetic topology of 1.8K NSP networks with a pattern of random

peering among them: NSPs originate routes for their own IP prefixes and propagate routes to authorized neighbors randomly. We also generated synthetic governance principals to delegate NSP security tags and customer IP prefixes from a common root. As in the previous experiments, the customer path control policies query NSP attributes (tags) endorsed from the authorities. We chose a tag delegation depth of 3 and IP prefix delegation depth of 3.

Figure 5.7 shows route authorization throughput as a function of route length for three sets of policies. There is a fixed cost to verify IP prefix ownership and validate routes, and an additional cost to check compliance with inbound and outbound path control policies at each NSP in the path. Thus the most expensive policy is **PBR-1** (Policy-Based Routing), which checks both inbound and outbound policies. **PBR-2** checks inbound path control policy only. We compare the results to basic BGPsec-like route validation and prefix ownership alone without customer-specified path control (labeled as **BGPsec**).

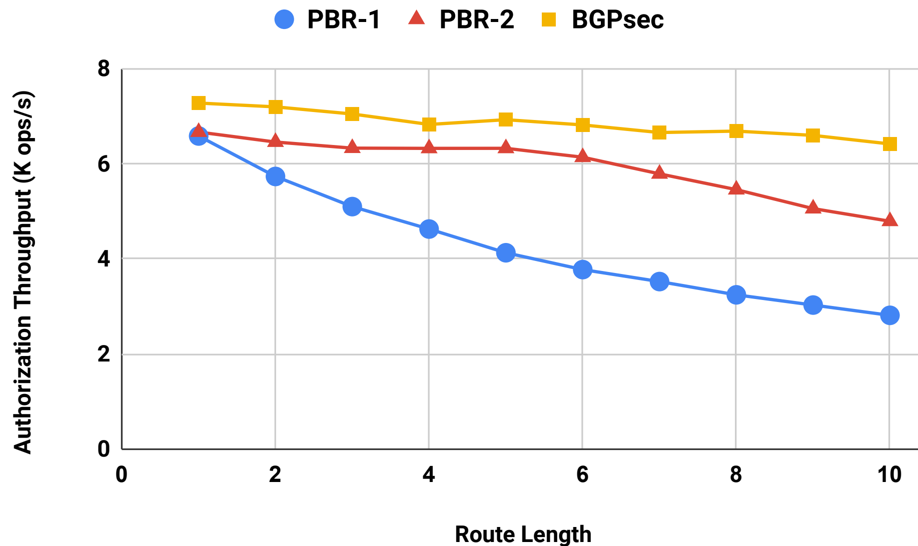



FIGURE 5.7: **SAFE logic engine throughput to validate routes with logical origin authentication and route attestation (equivalent to *BGPsec*), with inbound path control policies (*PBR-2*), and with both inbound and outbound path control (*PBR-1*).**

The results show that even for the most costly workload an NSP controller can check more than 2K routes per second. These checks occur only when the NSP receives a new

route or policy, and do not impact the dataplane. While Figure 5.7 does not include any crypto overhead (signature validation), these costs are fundamental for any routing security approach based on public-key cryptography (e.g., BGPsec). SAFE does impose additional costs to fetch linked certificates on demand, but the SAFE engine validates them once and caches their logic content until the TTL expires, which minimizes these costs for policies, governance endorsements, etc. (Figure 5.7 ran on a hot cache pre-warmed with all relevant certificates.) These results suggest that logical trust is fast enough to be practical at substantial scale.

### *5.4.4 Prefix Pair Matching with AQT*

We implemented AQT in Java that supports prefix pair updates and queries. We randomly generated IP prefixes with prefix length 8, 16 and 24, and IP prefixes with longer prefix length are children of IP prefixes with shorter prefix length. We randomly generate different numbers of prefix pairs from those IP prefixes and default prefix "0.0.0.0/0", with average prefix length about 23.8 and different average overlapping sizes. We insert, query and delete all prefix pairs with AQT and measure the performance with a single thread on a machine (Intel Xeon E5-2650 v2 @2.6 GHz). The result is shown in Fig 5.8. The results are consistent to the theoretical time complexity.

### *5.4.5 Dataplane Overhead for Source-Specific Routing*

We evaluate the overhead of source-specific routing with OpenVSwitch (OVS) [68]. OVS runs the virtual switches in the ExoGENI NSP deployments for our experiments. The exemplary policies for connectivity and path control in this dissertation require SDN routing rules that match packets on both source and destination. For example, with outbound path control policies, packets to a given destination $D$ might take different routes depending on the source $S$. This fine-grained policy control of routing may inflate routing tables and complicate packet classification at the SDN layer. The purpose of this section is to explain and quantify this cost, using the OVS software switch as a reference point.

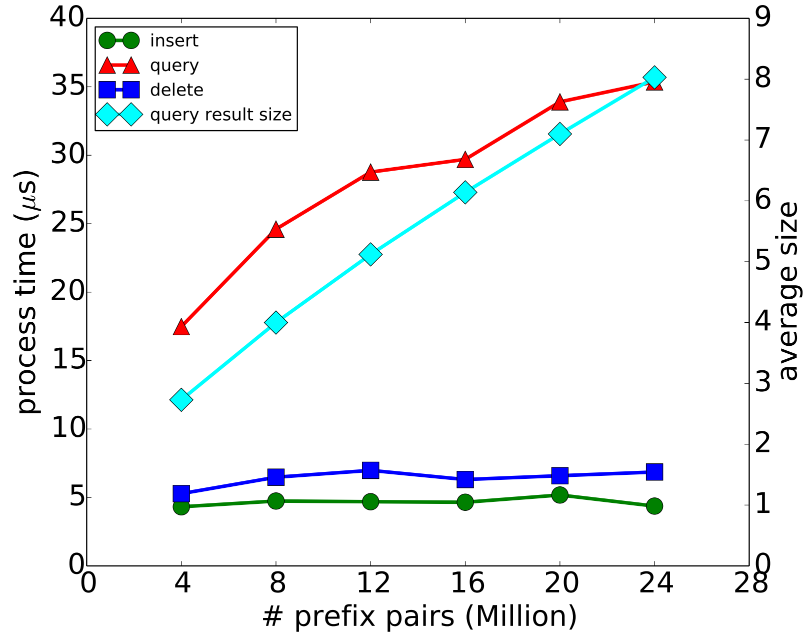



FIGURE 5.8: **Processing time for each prefix pair insertion, deletion and query operation with AQT. The time for insertion and deletion don't change much with the number of prefix pairs, as prefix lengths are about the same for all groups of prefix pairs. The time for query increases as the size of overlapped prefixes increases.**

OVS manages OpenFlow tables in userspace and a microflow cache and a megaflow cache in kernel. The megaflow cache is a single table of disjoint entries that caches the recent flows. The megaflow cache has a capacity of 200,000 flow entries in the latest mainstream OVS versions. The number of rules in megaflow cache are related with both the number of OpenFlow entries and the number of active flows. Packets in cached microflows or megaflows are fast processed in kernel. When the number of OpenFlow entries and active flows are both large, the megaflow cache will overflow and there will be cache misses. OVS will match the missed packet with OpenFlow tables in userspace, which is slow and expensive.

OVS classfies packets in both OpenFlow tables and the megaflow table with tuple space search. A tuple is a set of fields matched in the flow entries. Flow entries that match on the same tuple are put in the same hash table, where the keys are the hashes of the matched fields. The cost of tuple space search depends on the number of unique tuples (i.e. the number of hash tables) specified in the flow entries.

We evaluate the CPU overhead of packet classification with OpenFlow entries that match on both source and destination IP addresses in OVS. We create an ExoGENI slice with three "XO Extra Large" nodes in linear topology on the same rack working as source node, OVS node and sink node respectively. We run Open vSwitch 2.12.0 in Ubuntu 19.10 on the second node with 4 2.2 GHz cores (Intel(R) Xeon(R) CPU E5-2660 v2). We evaluate nine sets of policies. For the three sets of destination-based OpenFlow entries (*dst entries*), we randomly generate OpenFlow entries with the destination prefix within 32.0.0.0/8 and the length of the netmask between 8 and 24. The distribution of generated prefix lengths are subject to the IPv4 prefix cumulative distribution in [46]. There is also a default destination based OpenFlow entry that matches on "32.0.0.0/8" with lowest priority. For source-based OpenFlow entries (*src entries*), we randomly generate entries that match on both source and destination IPv4 prefix with source IP prefixes in 16.0.0.0/8 and the same destination IP prefixes as in *dst entries*. For each set of *dst entries*, we add $1x$ and $64x$ *src entries* additionally, resulting in six sets of *mixed entries*. We generate 8 *pcap* files with different number of tcp packets and with empty payload (66 Byte/packet) whose source and destination IP addresses are uniform randomly distributed in 16.0.0.0/8 and 32.0.0.0/8. The number of flows identified by source and destination IP address are about the same as the number of packets. We replay the traffic with *tcpreplay* at fixed rate 50K packets per second for multiple rounds and collect total CPU utilization of the OVS VM in a 10-minute period, and verify how much packets are received on sink node. The CPU utilization in the first round is higher than following rounds, as OVS needs to process every new flow with userspace OpenFlow tables. We omitted CPU utilization result in the first few rounds to illustrate the impact of source-specific routing for long-lived flows. We also sample and counter megaflow entries with *ovs-appctl* in seperate runs.

The CPU cost for packet classification are related with the number of OpenFlow entries, the type of the entries and the number of flows. Compared with *dst entries*, *src entries* leads to more packet classification cost due to two major factors. First, there are more hash tables for each packet to be matched against. Second, the number of flow entries could be

much larger with *src entries*. Given limited capacity, the megaflow cache for *src entries* overflows more easily and there will be much CPU cost in userspace.

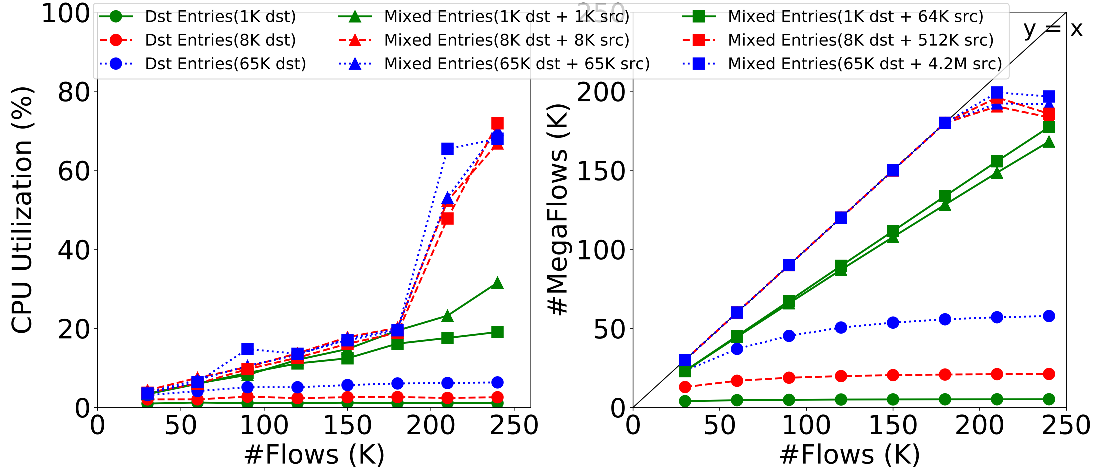



FIGURE 5.9: **The CPU utilization and the number of sampled megaflows with different numbers of flows and different numbers and types of OpenFlow entries. The packet loss rates are all less than 2%.**

Figure 5.9 shows the CPU utilization and the number of sampled megaflows with different numbers of flows and different numbers and types of OpenFlow entries. The CPU utilization with *mixed entries* are higher than that with *dst entries*. With a small number of flows or a small number of OpenFlow entries, the packets are processed in fast path with the cached microflows and megaflows in kernel for both *dst entries* and *mixed entries*. And the CPU utilization for *mixed entries* are about $1x$ to $20x$ higher than that for *dst entries* with less than 200,000 flows. When the megaflow cache is not full, the number of OpenFlow entries doesn't affect the CPU utilization much, as the classification cost mainly depends on the number of unique tuples in megaflow table. However, the number of OpenFlow entries still matters in the way that the number of megaflows with more OpenFlow entries are also larger. The megaflow cache for more larger OpenFlow entries will overflow earlier as the number of flows increase, as we can see from the Figure 5.9. With more than 200,000 flows and a large number of *mixed entries*, the megaflow cache overflows and there are much more processing overhead in userspace. The CPU utilization with *mixed entries* could be more than $50x$ higher. In our experiment, the maximum number of megaflows for *dst entries* are

capped at $6.5K$ as the destination IP address are sampled from a relatively small prefix 32.0.0.0/8. Therefore, we don't see the megaflow cache for *dst entries* overflowing, and the CPU utilization for *dst entries* are low. But with *dst entries* specified in a larger address space, we also expect to see the performance drop with a large number of OpenFlow entries and flows.

# 6

# Secure Customizable Internet Filtering

## 6.1 Introduction

Route filtering and ingress filtering are important approaches to addressing the threats in the Internet including route attacks and IP address spoofing. Ingress filtering based on uRPF (§6.2) is automatic, but its effectiveness relies on the effectiveness of route filtering. The key challenge to defending against spoofing and hijacking is to determine whether any given imported route is "legitimate" before accepting it.

As a starting point, there are security standards that use signed certifications to authenticate routes and prevent certain kinds of route forging (BGPsec [80]), and to validate that routes originate with the legitimate owners of address prefixes (RPKI [21]). Though these protocols are not yet fully deployed, their behavior and limitations are well-understood. These defenses are not sufficient (e.g., see [40]). By malice or misconfiguration, networks may export routes that are valid (i.e., the exporter has a certified path to the source) but are outside of their intended scope [81]. That means the path export is not compliant with the route export policies of some predecessor in the path. These *route leaks* may permit the exporter to source or sink traffic that it is not trusted to carry. (See §6.2.3.) By August 2020, QRATOR Labs [71] has reported 8 cases of route leaks in the year affecting massive

number of prefixes and ASNs.

A more complete defense requires the network domains to share policies for route export/import. These domains are autonomous systems (AS) in the Internet; as throughout this dissertation, we refer to the participants in inter-domain networking generically as network service providers (NSPs). Policy sharing among NSPs permits them to identify routes that are *faithful* to all relevant policies, and filter routes that are unfaithful, i.e., are not compliant with policies of predecessor NSPs in a path. The leading example of this approach is MANRS [3], which promotes sharing of route import/export policies through the Internet Routing Registry (RADb) [72].

We can understand route legitimacy from the perspective of trust. As a simple example, when a provider P exports a route R to a customer C, P trusts C to use R for traffic sourced from C's networks and descendent customers (C's *customer cone*). But P does not trust C to export R to another neighbor N and use R for their traffic. For example, that could permit the untrusted C to spoof traffic from the source of the route into N, or capture return traffic.

Building on this idea, we define a trust predicate that captures whether a given NSP is *eligible* to carry traffic between a given (source, destination) prefix pair. The important challenges then are how to specify the policies of the customers and NSPs that bear on eligibility, supply an inference procedure to evaluate eligibility, and identify relevant policies as input to the procedure.

It is worth noting that anti-spoofing approaches with ingress filtering in the control plane rely on trust in eligible NSPs not to accept/inject non-compliant packets that may be spoofed. Once the traffic enters the network at an eligible NSP, it is no longer possible to block it. To obtain source authentication that is stronger than possible with route filtering alone, researchers have developed various approaches that integrate cryptographic authentication into the packet headers and routers. (See Chapter 2.) But even those rely on the routers not to leak their keys.

The customer networks could specify their own routing policies about what NSPs are

trusted for their traffic. Customer networks can also delegate their routing policies to trusted parties like trusted NSPs or network authorities. Enabling customers to choose which NSPs to trust for their traffic is also in line with the "design for choice" principle to ease the tussle in the Internet [28].

In this chapter, we propose to define the routing policies of customers and NSPs rigorously in a logic language (Datalog) with SAFE [22] logical trust framework. We propose Secure Customizable Internet Filtering (SCIF) that extends the path control policy in chapter 5 to limit the eligibility of NSPs. SCIFis a logic package that is directly deployable with ExoPlex in testbed-hosted NSPs. With proper routing policies of the customer networks and NSPs, SCIF protects the route propagation thus enabling effective uRPF based ingress filtering. We survey existing approaches for routing security and anti-spoofing and show how to represent and implement existing approaches with SCIF in the control plane (§6.3). SCIF allows customer networks to specify customized routing policies to endorse NSPs for their traffic (§6.3.2). Alternatively, SCIF allows routing policy and privilege delegation to capture the NSPs' routing policies with their neighbors (§6.3.3). And SCIF supports specifying routing policies with groups of objects or groups of principals for scalability (§6.3.4). Comparing existing approaches with our customized routing policies also helps us to understand their strengths and limitations and possible improvements (§6.3.5, §6.3.6). Lastly, we evaluate the performance of logical trust for routing policies of NSPs and show the applicability of logical trust for routing policies at the Internet scale (§6.4).

## 6.2 Overview

An interdomain network comprises a collection of NSPs that exchange traffic to provide end-to-end connectivity for customer networks. NSPs in the network may be controlled by different parties. NSPs must defend against neighbors that behave unexpectedly or unfaithfully due to malice or misconfiguration. An NSP controls its dataplane to route traffic and exchange packets with other NSPs over links that connect their networks at layer 2. NSPs also exchange route and policy information with other NSPs in the control

plane. The term NSP encompasses autonomous systems (AS) in the Internet.

Source accountability is a property of a network: the receiver of a packet can always identify its source. A variety of techniques can provide it by tagging packets with cryptographic identifiers and integrating cryptographic checks into the dataplane. The Internet architecture does not mandate any such checks: packets with spoofed source addresses could appear on any untrusted external link. Source accountability is present only to the extent that NSPs filter (discard) any spoofed packets arriving at their edges. If any unfaithful NSP injects or admits a spoofed packet, it may pass that packet to any upstream NSP that trusts it to carry that packet to its own edge, and accepts the packet on that basis. Once it enters the network, such a spoofed packet may reach its destination.

But how should NSPs decide which traffic to filter? Lacking an ability to verify packets in the dataplane, it is necessary to distinguish legitimate traffic from unexpected suspect traffic based on rules known in the control plane. Traffic is legitimate if it arrives from a source that is eligible to carry it based on prefix ownership and the delegations of trust implicit in route advertisements. For example, any edge subnet is trusted to source traffic from any prefix that it owns. Similarly, an edge subnet trusts its provider NSP to carry its traffic to any destination, and to deliver traffic from any external source. The provider NSP delegates eligibility for subsets of this traffic by advertising routes for the subnet to upstream NSPs.

This chapter explores control-plane approaches to defend against spoofing. Their effectiveness depends on how precisely the NSPs can distinguish legitimate traffic. They include IETF Best Common Practice (BCP) recommendations and security protocols in various Internet RFCs. We consider how to enhance current defenses in two directions. First, we consider more expressive route advertisements that capture their intent, enabling downstream NSPs that receive derived routes to validate their compliance. Second, we consider path control policies that constrain the set of NSPs that are trusted for specified traffic. We use trust logic to express these advertisements and policies and check compliance automatically.

We presume that NSPs are able to exchange control-plane messages with their neighbors. We do not specify how they establish connectivity: the Internet Border Gateway Protocol (BGP) offers a solution, and testbed-hosted NSPs may use testbed control networks. We also presume that NSPs sign control plane messages under keypairs, and subscribe to a common set of governance authorities, as described elsewhere in this dissertation. The authorities act as trust anchors for secure prefix ownership in a common address space. They can also bind prefixes and NSP keypairs to identities (or attributes of identities) in the real world where required. For example, the Internet has the root authorities for RPKI (prefix ownership) and DNSSEC (registration). We can use trust logic to provide similar governance structures for testbed-hosted NSPs.

### *6.2.1 Internet Defenses for Spoofed Traffic*

BCP 38 [36] mandates edge providers to configure ingress ACLs that restrict ingress traffic from downstream networks to known or advertised prefixes. BCP 38 targeted networks close to the edge where the list of advertised prefixes is relatively static. It presumed manual ACLs, which are vulnerable to misconfiguration.

BCP 84 [9] introduces automated ingress filtering with Reverse Path Forwarding (RPF). With unicast routing it is called unicast RPF (uRPF) to distinguish it from a similar approach used in multicast protocols. With uRPF, an NSP applies ingress filtering rules for incoming packets according to the routes that it accepts (see Chapter 2). In this way, ingress filtering is tied to route filtering: an NSP must apply a control policy to filter its received routes and reject non-compliant routes. If a neighbor advertises a route, an NSP must determine if the route advertisement is legitimate before accepting it. If it accepts (or selects) the route to a prefix $P$ from a neighbor, then it also trusts the neighbor to pass traffic sourced from $P$. Since it also directs traffic destined for $P$ to that neighbor, accepting the route permits the neighbor to hijack traffic destined for $P$ as well. If an NSP can determine which routes to accept as faithful and legitimate, then traffic hijacking is defeated and ingress filtering of spoofed traffic with uRPF is automatic.

But which routes are legitimate? As a starting point, a legitimate route advertisement must be valid: it must be originated from an NSP authorized by the prefix owner (ROA) and the propagation of the route is certified hop by hop, as required by RPKI [21] and BGPsec [80] (see Chapter 2).

However, not all valid routes are legitimate. If a destination prefix $P$ is reachable to an NSP, then that NSP can export the route to a neighbor: BGPsec considers the resulting path to $P$ as valid. In this way, any malicious or misconfigured NSP can propagate an illegitimate path and place itself on a feasible path to $P$.

Even if a route advertisement is valid according to BGPsec, there is no guarantee that the reverse path is feasible for traffic in the reverse direction. Origin authentication and route validation protect routes from being forged, but an NSP can still propagate a valid route outside of its intended scope. For example, in Figure 6.1, a provider NSP $A$ advertises a route to an external prefix into a multihomed customer network $B$. $A$ trusts $B$ to propagate the route to its customer networks (if any), but not to carry traffic between $A$ and another provider or peer. Suppose $B$ leaks the route to another provider NSP $C$. If $C$ accepts the route, then $B$ can hijack traffic from $C$ to $A$ and to spoof traffic from $A$ to $C$, which it should not be empowered to do.

A *route leak* is the propagation of a route to a scope that was unintended by a preceding NSP in the path. A malicious NSP could intentionally leak routes by manipulating its route import and export policies [40]. It could advertise a path for the source to a colluding neighbor NSP without proper route filters to make itself on a path for uRPF. The route leak enables it to spoof traffic from a leaked prefix or attract traffic destined for it. For example, a multihomed network could leak the routes it learned from one provider network to another provider. While the route is valid by BGPsec, it may not be a faithful reverse path.

MANRS [3] also recommends uRPF for anti-spoofing. To enable global route validation against the routing policies of NSPs in the path, the Internet Routing Registry (RADb) [72] allows NSPs to register and store their routing objects and routing policies in a cooperatively

maintained distributed database. RPSL [6, 56] is the policy language that allows NSPs to specify their import routing policies and export routing polices. NSPs can pull the registry information of other NSPs for global route validation and filtering.

However, RPSL does not support coordination between NSPs: one NSP cannot affect how routes are propagated beyond its neighbors with its routing policies. And the registry approach has certain limitations. In today's RADb, the policy information is voluntary, incomplete, generally unauthenticated, and sometimes incorrect. Secure RPSL [56] verifies that changes to objects in the database are done by the legitimate holders of the Internet resources mentioned in those objects. But there is no auditing or authorization for the NSPs' routing policies, as the business relationships and agreements between NSPs may be confidential. While compliant to routing policies, a route can still be propagated out of the intended scope of the preceding NSPs in the path if NSPs intentionally manipulate their routing policies.

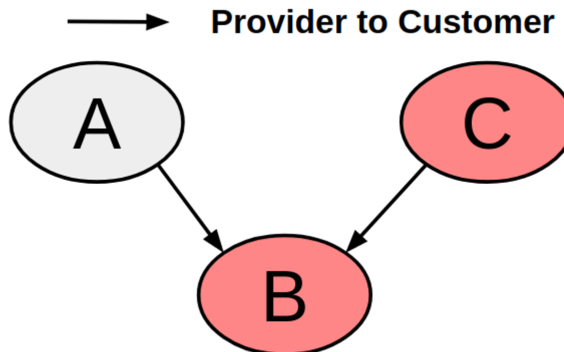



FIGURE 6.1: **An example route leak: a multi-homed network $B$ leaks the route it learned from its provider $A$ to another provider $C$.**

**Malicious/misconfigured NSP.** Any malicious or misconfigured NSP (without proper ingress filtering) on the route accepted for uRPF can be a vector for spoofed traffic.

**Colluding attack.** Proper route import and export policies (§6.2.2) of NSPs would prevent route leaks by single point of failure. If an NSP leaks a route maliciously or by mistake, its neighbor would be able to detect and filter the leaked route with its import policies. But colluding neighbors can still leak routes by hiding their real business relationship and

specifying their import and export policies to allow such route leaks.

We need a more rigorous notion of legitimacy for routes, based on the routing policies and relationships of the NSPs on the path.

### *6.2.2 Routing Policies of NSPs*

We focus on what information is required for route filtering, and show how to propagate it safely using logical trust. We show how customer networks and NSPs can expose restrictions on what NSPs can carry their traffic and how their route advertisements propagate.

### *6.2.3 Limiting the Adversary: Trust Model*

**Eligible NSP.** When an NSP determines which route to accept for uRPF based ingress filtering, it is important that all NSPs in the route are eligible for the traffic.

We formalize the notion as follows. We say that an NSP is *eligible* for a prefix pair if both the source and the destination trust the NSP to carry the traffic. Equivalently, we can say each NSP has an *eligible set* of prefix pairs for which it is eligible to carry traffic.

The eligibility relation for each NSP must be a relation on prefix *pairs*. For example, everyone must trust an NSP to source traffic from its customers and route traffic to its customers, i.e., to route traffic from its own sources to any destination, or from any source to its own destinations. But that does not imply trust in the NSP to carry traffic from any source to any destination.

**Symmetric trust.** For simplicity, we initially assume that the trust relation is symmetric: customer networks trust the same NSPs to carry their traffic in both directions. But it is not necessarily symmetric and we distinguish between policies for inbound and outbound trust.

**Faithful routes.** A "faithful" route is also defined over prefix pairs. A *faithful* route is valid and compliant to the routing policies of NSPs on the path and the routing policies of the source and destination: all NSPs in the path are eligible for the prefix pair. A route might be faithful for some prefix pairs, but unfaithful for others.

The set of faithful routes is also related to the set of valley-free routes [37], but they are not the same. Valley-free routes are not necessarily faithful and vice versa. Many works have attempted to preclude valley routes on the implicit presumption that such routes are illegitimate. On its face, it seems to be a reasonable presumption. However, it has been shown that valley announcements are common and flow from the policies of the NSPs involved [39]. The concept of valleys and valley-free routing rests on classifying the relationships among neighboring NSPs (e.g., customer-provider). It seems that these relationships are not absolute, but rather are relative to specific prefix pairs.

Each NSP may filter traffic arriving over unfaithful paths. Each NSP on a faithful path is trusted by the source and destination to filter traffic in a compliant way, i.e., to forward a packet for the prefix pair only if it arrives from an interface that is on a faithful path. If the NSP is untrustworthy, it can be a vector for spoofed traffic for the prefix pair in its eligible set. We cannot stop an NSP from being a vector for spoofed traffic for prefix pairs in its eligible set: we must trust it for those prefix pairs. But we can block the NSP from spoofing traffic for prefix pairs outside of its eligible set.

### *6.2.4 Logical Routing Security*

We use logic to express NSP's eligibility for prefix pairs and accommodate flexibility for customized routing policies. Customer networks and NSPs can control how their routes are propagated by restricting the eligibility of NSPs for their traffic with privilege delegations.

We use SAFE [22] logical trust framework to manage routing policies in a logic language (Datalog). It signs and stores logic statements in a decentralized certificate repository, and answers logical queries with a logic engine (Styla). SAFE simplifies the discovery and retrieval of the logic content relevant to a trust decision with *certificate linking*, which is important for efficiency for interdomain networking in the scale of Internet.

The logical trust approach is a rapid prototyping vehicle for experimental approaches to secure routing. We can express various existing approaches precisely and rigorously in logic, which allows to understand and compare the strength of different approaches. It

also makes easy to explore the design space just by modifying the logic policies within the same routing context. The logical routing policies are directly deployable with augmented routing specifiers and rules on testbed-hosted NSPs using ExoPlex controllers (see §5.2). Experimenting with different network policies in the testbeds helps us to understand the strengths and limitations of different approaches in the Internet.

## 6.3 SAFE Customizable Internet Filtering

SCIF aims to secure routing that allows routes to be propagated to trusted NSPs and intended scope so that NSPs can construct effective ingress filters for anti-spoofing automatically based on uRPF. With uRPF, we allow only traffic for prefix pairs from interfaces that are on the faithful routes for the accepted prefix pairs. Then we need proper spoofing control policies to control how route advertisements are propagated and determine what routes are faithful.

SCIF incorporates builtin secure routing policies in ExoPlex for route origin authentication and route validation that resemble RPKI and BGPsec, with additional customized routing policies to contain the untrustworthy NSP to be eligible for only specific prefix pairs (e.g. between its own subnets to other subnets). By limiting the eligibility set of an NSP with customized routing policies, we contain prefix pairs that the NSP could forge traffic for and also prevent the route from being leaked for ineligible prefix pairs.

The customer network can directly endorse NSPs for their traffic (§6.3.2). Or the customer network delegates its routing policies to trusted parties, and the trusted parties can redelegate the privilege to other NSPs (§6.3.3). Delegation is the process that the delegator endorses the delegatee with privilege for objects which the delegator is privileged for. In SAFE, the delegator signs a certificate that includes the endorsing statement and links for credentials that prove the delegator's privilege. Then the delegatee can prove its privilege for the delegated objects to others by presenting the certificate.

SCIF provides a common formalism for different policies and different approaches, including valley-free routing, RPSL and customized policies with stronger security.

As a starting point, we consider the straw man: When advertising a route to an untrustworthy NSP, we can specify if the route is delegatable/propagatable. In the example of Figure 6.1, when A advertises a route to B, it makes the route undelegatable. We can equivalently express such policy with SCIF that the untrustworthy NSP B is only eligible for traffic between its own prefixes and other subnets. When B leaks the route learned from A to C, C would determine that B is not eligible for any traffic that C would forward to B, thus not accepting or propagating the route. But this only works at the edge of the network.

Another approach is valley-free routing. [70] proposes to label different routes following a state transition model that changes the state of the route according to the current state and the nature of the neighboring relationship. We can encode the labeling idea neatly in logic, and the result excludes exactly the routes that the algorithm in this reference is designed to exclude. However, it is vulnerable to certain scenarios of NSPs who falsify the nature of an announcement, forcing downstream neighbors to accept a risk of spoofing. (But this misbehavior is accountable.) [70] summarized four types of route leaks that violate the valley-free model, when the NSP is supposed to export the route only to its customer networks but leaks the route to its peer or provider. With SCIF we can also express valley-free policies in that each NSP is eligible only for traffic between its customer cone and other subnets.

SCIF captures existing approaches like RPKI/BGPsec, routing policies of NSPs and allows for customized and more selective routing policies to reject untrusted NSPs. A comparison of sets of accepted route advertisements by BGPsec, RPSL, MANRS and more restrictive SCIF policies is shown in Figure 6.2.

### *6.3.1 Capturing RPSL with Logic*

RPSL allows a network operator to express local routing policies with route filters built with prefixes, AS sets or route sets. The declaration of AS sets and route sets can be nested, i.e., an AS set can have another AS set as its member, and a route set can have another

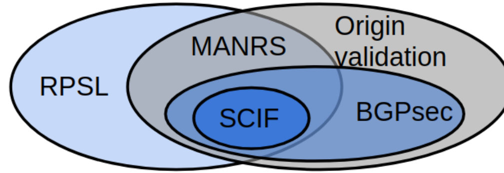



FIGURE 6.2: **Accepted routes by BGPsec, RPSL, MANRS and more restrictive SCIF policies. BGPsec accepts routes with certified ROAs (origin validation) and authenticated route delegations. RPSL requires route validation with NSPs' registered routing policies in RADb. MANRS recommends both origin validation and route filtering with routing policies in RPSL. In addition to BGPsec and routing policies of NSPs, the more restrictive SCIF policies require additional path compliance checks against customized routing policies that qualify eligible NSPs, thus accepting a strict subset of those accepted by MANRS.**

route set or AS set as its member. Those policies and objects are stored in a cooperatively maintained distributed database (RADb) to provide a global overview of the routing policies of the ASes. RPSL supports both IPv4 and IPv6: this chapter focuses on routing policies for IPv4, but the ideas are applicable to IPv6 as well.

The basic forms of the import and export routing policies are "from $\langle peer \rangle$ accept $\langle filter \rangle$" and "to $\langle peer \rangle$ announce $\langle filter \rangle$". The *peer* can be expressed as AS number or AS set. The *filter* can be expressed as AS number, AS set, route Set, explicit set of prefixes or regular expression. Those routing policies are generally specified according to the relationship with the neighboring AS. For example, a provider AS may accept only routes that originate from a customer AS, while a customer AS may accept any routes from its provider AS. We show this can be captured in logic in §6.3.1.

Routing policies in RPSL specify what routes to import and export from neighbors. We can capture those routing policies in RPSL with logic. STRONG group service [22] supports specification of groups of principals or objects and membership checking. STRONG groups can also be nested, similar to AS set and route set in RPSL. Listing 6.1 shows an exemplary import routing policy in logic that accepts a route from a neighbor AS if the neighbor AS and the route match with an import ACL entry. Export routing policies are similar. We ignore certain features of RPSL: route filtering by prefix length and by path patterns in regular expressions.

**Listing 6.1: An example route import policy in SAFE logic.**

```
accept(?Neighbor, ?Route):-
  importAcl(?ASset, ?RouteSet),
  membership(?ASset, ?Neighbor),
  membership(?RouteSet, ?Route).
```

### *6.3.2 Customer Routing Policy*

Customer networks can endorse NSPs for their traffic with customized routing policies. An NSP is eligible for a prefix pair iff both the source and the destination trust the NSP to carry traffic for the prefix pair (Listing 6.2). Listing 6.3 shows the template script for a prefix owner to endorse an NSP for its traffic.

**Listing 6.2: Logical policy to authorize an AS's eligibility for a prefix pair. <: is a builtin that asserts the prefix containment.**

```
eligible(?NSP, ?PrefixA, ?OwnerA, ?PrefixB, ?OwnerB) :-
  ?OwnerA: ownPrefix(?PrefixA),
  ?OwnerA: trust(?NSP, ?PrefixA, ?PeerPrefixA, _),
  ?PrefixB <: ?PeerPrefixA,
  ?OwnerB: ownPrefix(?PrefixB),
  ?OwnerB: trust(?NSP, ?PrefixB, ?PeerPrefixB, _),
  ?PrefixA <: ?PeerPrefixB.
```

**Listing 6.3: A template script for a prefix owner to delegate routing privilege to anther principal with a link to the certificate proving that the issuer owns the IP prefix.**

```
defcon delegatePrivilege(?NSP,?Prefix, ?PeerPrefix,
  ?Delegatable, ?Cert) :-
  {
    link($Cert).
    trust(?NSP, $Prefix, ?PeerPrefix, ?Delegatable).
  }.
```

The customer network may trust an NSP to carry its traffic for different reasons including but not limited to the following ones: (1). It may trust its provider NSP to carry traffic between its subnet and any other subnet. (2). The customer network can also base its trust on some governance structure. For example, a customer network may trust an NSP if an trusted authority asserts that the NSP is implementing proper route filtering and ingress filtering and not spoofing any traffic. The customer can also specify attribute-based policies that trust an NSP if the trusted authority endorses the NSP with certain attributes. (3). The customer network can also base its trust on its knowledge of the network topology and business relationships of NSPs. For example, if an NSP is near the edge, the customer

network may trust the NSP to carry traffic between its subnet and all subnets in the NSP's customer cone.

### *6.3.3 Routing Policy Delegation*

When customer networks are not able to discriminate which NSPs are trusted for which prefix pairs—or they do not want to manage their routing policies—they can delegate their routing policies and routing privileges to trusted NSPs (Listing 6.4). For example, the customer network A can specify policies that their provider and core transit NSPs are trusted to carry traffic between its subnet and all the other subnets and allows them to delegate the routing privileges. When the core transit NSP B propagates the route to its customer NSP C, B can specify routing policy that C is trusted to carry traffic between A and a set of prefixes for which B will choose C as next hop. For example, if C is a customer NSP of B, B may delegate to C to carry traffic between A's subnet and C's customer cone (all subnets that C can reach via customer links). This makes C eligible only for prefix pair between A and C's customer cone. This limits C from placing itself on a feasible path between A and any other destination than C and the customer networks of C.

**Listing 6.4: Customer routing policy delegation: A customer network delegates their routing policies to trusted NSPs, allowing trusted NSPs to delegate routing privileges for more specific prefix pairs.**

```
trust(?NSP, ?Prefix, ?PeerPrefix, ?Delegatable) :-
  ?Delegator: trust(?NSP, ?Prefix, ?PeerPrefix,
    ?Delegatable),
  trust(?Delegator, ?Prefix, ?SupPeerPrefix, true),
  ?PeerPrefix <: ?SupPeerPrefix.
```

SCIF with policy delegation can provide an equivalent level of security as RPSL. And it is as easy for NSPs to make such delegations as to specify routing policies in RPSL. In the import policies of RPSL, an NSP specifies what routes/prefixes are accepted from a neighbor or a group of neighbors. We can implement equivalent import policies with SCIF that trust the neighbor to carry traffic only between the accepted prefixes and any other prefixes. The customer network can directly specify such policies or trust the NSP to delegate the routing privileges to the neighbor. When an NSP receives from a neighbor a route that is rejected by RPSL, it will also reject the route with SCIF since the neighbor is

not eligible for any traffic between the advertised prefix and any other prefix.

Note that NSPs can control how their routes are propagated by routing privilege delegations to their neighbors, but the trust is rooted in the owners of the source and the destination. That is, the owners of the source and destination subnet need to delegate their routing policies and privileges for the prefix pair to the trusted NSPs when they are not sure if certain other NSPs can be trusted.

### *6.3.4 Routing Policy with Groups*

Instead of exhaustively listing IP prefixes and AS numbers, we can define routing policies with respect to groups of NSPs and groups of IP prefixes that are similar to “AS set” and “route set” in RPSL [6]. And groups can be nested. For example, a provider NSP can create a group for its customer subnets, and delegate the group privilege to its customers' groups.

There are import policies in RADb that accept only limited route sets from certain neighbors. This indicates that the NSP will forward only traffic to those route sets via those neighbors. Accordingly in SCIF, the NSP can delegate to those neighbors only the routing privileges for traffic sourced from or destined to the route sets. Listing 6.5 shows the logical policy that verifies if an NSP is trusted for a prefix pair given delegation with group. When answering such a query, we check if the delegator is eligible to delegate the queried prefix pair. Since Datalog doesn't support query at set level (i.e., query if all members of a set are members of another set), we can not determine the delegator's eligibility for all the delegated prefix pairs. But the policy is sound and safe to determine the delegator's eligibility for every queried prefix pair.

Similarly, an NSP can delegate routing policies to an NSP group. NSPs can also create new groups to aggregate their eligibility set that are delegated from different sources. Specifying routing policies with groups improves the scalability of SCIF. The certificates for delegations can be linked in such a way that SAFE server can pull the needed certificates following the links when authorizing a query. And those principals are free to optimize how

the certificates are linked in different scenarios.

**Listing 6.5: Logic rules that authorize the eligibility for a prefix pair with privilege delegations in terms of route set. An AS is eligible for a prefix pair if the prefix pair are members of the delegated route set, and the delegator is also eligible for the prefix pair with delegation privilege.**

```
trust(?NSP, ?Prefix, ?PeerPrefix, ?Delegatable) :-
  ?Delegator: trust(?NSP, ?RouteSet, ?PeerRouteSet,
    ?Delegatable),
  membership(?RouteSet, ?Prefix),
  membership(?PeerRouteSet, ?PeerPrefix),
  trust(?Delegator, ?Prefix, ?PeerPrefix, true).
```

### *6.3.5 Expressing MANRS with SCIF*

MANRS recommends route filtering with origin validation and RPSL, and ingress filtering based on uRPF. This can be viewed as a special case if we specify SCIF policies as follows: An NSP is trusted to carry traffic from a prefix if the NSP has a route to the source prefix that is origin validated and compliant to local routing policies of all preceding NSPs.

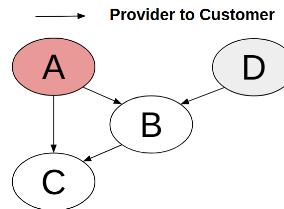



FIGURE 6.3: **NSPs must prefer routes learned from customers over routes learned from peers/providers for ingress filtering. If NSP B fails to do so, A will still be able to spoof traffic from C to D via B, even though NSPs' routing policies ensure valley-free routing and A is not on a valley-free route from C to D.**

The effectiveness of ingress filtering in MANRS then depends on the routing policies of the NSPs. If the routing policies of NSPs allow only valley-free routes and NSPs always prefer routes learned from customer NPSs over those learned from peers and provider NSPs, any NSP that is not in a valley-free route between the source and destination is not able to spoof traffic for the prefix pair.

The importance of preferring routes learned from customer NSPs for ingress filtering given valley-free routing is shown in Figure 6.3. In general, if a downstream NSP accepts one route for a prefix that contains an upstream NSP, it is important that all routes that

the upstream NSP accepts for the prefix are also compliant to the routing policies of the downstream NSP. Otherwise, NSPs in any route that is accepted by an upstream NSP but not by the downstream NSP would be able to inject spoofed traffic sourced from the advertised prefix via the upstream NSP.

### *6.3.6 Improving MANRS*

To improve the anti-spoofing security provided by MANRS, we recommend more secure and strict route filtering measures as follows.

First, use BGPsec instead of origin validation to validate a route.

Second, introduce trust and authorization for routing policies of NSPs and the membership of AS sets (customer cones) and route sets. It requires honest and authorized local routing policies and set membership to prevent route leaks. QRATOR Lab [71] reported an route leak event on August 24, 2020 that the receiving NSP is unable to filter the leaked routes with AS-set based filter because of misconfigured AS set by the leaker NSP. To authorize routing policies and set membership in a multidomain environment, it is important to root the trust in some governance that all parties agree on instead of the maintainers/owners of the NSPs.

Third, prefer uRPF strict mode over feasible mode when possible. As in the example of Figure 6.3, a route that is accepted by an upstream NSP is not necessarily accepted by downstream NSPs. NSPs in the routes that are not accepted by downstream NSPs may inject spoofed traffic via an NSP in the accepted route if the NSP implements uRPF in feasible mode.

Last, given unauthorized routing policies in RADb, it is possible to prevent certain route leaks from colluding neighbors in some scenarios by validating the reverse path with those routing policies. When receiving a route from a neighbor, the NSP checks if any route that is exported by itself to the neighbor can be propagated to the source subnet. For example, consider in Figure 6.1 that B and C collude to leak routes learned from A. A will block routes from B for other prefixes than those directly owned by B with proper route filters.

Thus another NSP D who receives the leaked route from C, would be able to determine that the route is leaked if all routes that D exported to C are denied by A.

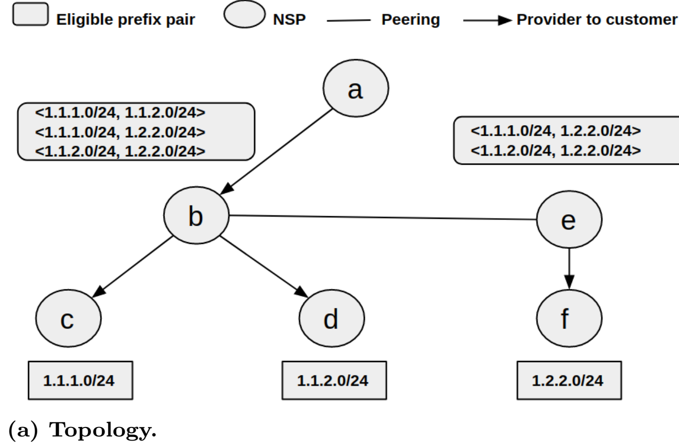
### *6.3.7 Route Propagation and Ingress Filtering*

As in [92], when receiving a route advertisement, the NSP would compute for all prefix pairs that this route advertisement is compliant. If the route is faithful for some prefix pairs that the NSP would choose for the traffic, the NSP can install acceptlisted ingress filtering rules and propagate the route to trusted neighbors. And an NSP will not propagate the route is it is not using the route for any traffic.

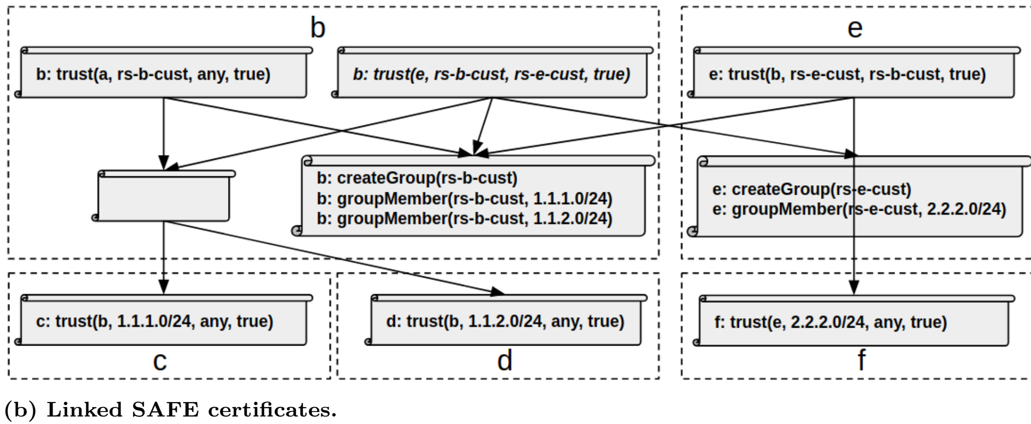
Combining SCIF and ingress filtering between ASes, we can limit NSPs from spoofing traffic if they are not on a faithful route for the prefix pair. Our approach works in the control plane and requires neither packet verification in the dataplane nor upgrades to routers.

Figure 6.4 shows an exemplary NSP topology with routing privilege delegations in terms of groups. Figure 6.4a shows the NSP topology with customer-provider and peering relationships. The customer NSPs delegates the privilege for prefix pairs between their customer cones to all other prefixes to their provider NSPs. The NSP delegates to the peer NSP the privilege for prefix pairs between its customer cone and the peer's customer cone. They also create groups for their customer subnets. NSP $b$ also issues a new certificate that aggregates the links for certificates issued by its customers. With those delegations in Figure 6.4b, NSP $b$ is eligible for prefix pairs between its customer subnets and the prefix pairs between its customer subnets and NSP $e$'s customer subnets. NSP $e$ is eligible for prefix pairs between its customer subnets and $b$'s customer subnets.

In this example, NSP $b$ is not able to leak the route from NSP $e$ via NSP $a$. When the Exoplex controller of NSP $a$ receives the route from NSP $b$ for the customer cone of NSP $e$ , it will authorize eligibility set of NSP $b$ with SAFE logic engine. NSP $a$ will not propagate the route because NSP $b$ is not eligible for traffic between the customer cone of NSP $e$ and any other networks than the customer cone of NSP $b$. Accordingly, NSP $a$ will




(a) Topology.




(b) Linked SAFE certificates.

FIGURE 6.4: **An exemplary customized routing policy with privilege delegation: customer networks $c$, $d$ and $f$ trust their provider NSPs to delegate their routing privileges. NSP $b$ and $e$ create groups for their customer subnets. NSP $b$ and $e$ peer with each other and delegate routing privileges for prefix pairs between their customer subnets to each other.**

not accept any traffic from NSP $b$ that is not in the eligibility set of NSP $b$. Thus NSP $b$ is limited from spoofing traffic from NSP $e$'s customer subnets to any other destinations than its customer subnets.

## 6.4 Experiments

We want to evaluate the performance of logical trust for SCIF with delegations in Internet scale. How NSPs delegate routing privileges for their neighbors depends on their relationships in similar way to their routing policies in RPSL. RPSL allows NSPs to express their

local routing policies with route sets and AS sets. We would also define nested NSP groups and prefix groups similarly when specifying and delegating routing policies.

The performance of logical trust for SCIF with delegations are related with how NSP groups and prefix groups are specified, including their sizes and the depths of nested groups. While the performance of logical trust for SCIF depends on the complexity of policies, we can estimate the cost by implementing those routing policies in RPSL from RADb with SAFE logical trust and evaluating the performance of logical trust for NSPs' routing policies.

We downloaded the dateset from RADb, with about $60K$ ASes, $1.5M$ prefixes, $25K$ AS sets and $6.5K$ route sets maintained by $26K$ maintainers. We assign key pairs to AS owners and maintainers for AS sets and route sets, and synthesize trust roots for prefix allocations. As the propagation of route advertisements for different prefixes are independent, we randomly sampled 800 prefixes and propagate the route advertisements for them through the whole AS network when allowed by the import and export policies of ASes. In the experiment, we prefer the shortest path.

We run a SAFE instance on a 48-core KVM (Intel Xeon CPU E5-2670 v3 @ 2.30GHz ) with 126 GB of RAM and 10 Gb/s Ethernet. We evaluate the authorization throughput for import and export queries on cached contexts (fetched and validated logic statements).

After the routing table became stable, the average length of paths was about 4.7. About 470 routes (prefixes) reached more than $39K$ ASes ($42K$ ASes on average) and about 290 routes reached less than 10 ASes. The missing export or import policies is the main cause for limited propagation of those routes. The average authorization throughput for import queries is about $4.75K$ ops/s (operations per second), and the average throughput for export queries is about $4.68K$ ops/s. The result shows that logical authorizations for routing policies of NSPs are fast given the use of groups in the Internet scale. It also indicates that logical trust for SCIF with delegations will be fast enough to be deployed in practice.

# 7

# Discussion

## 7.1 Datalog for Network Policies

Datalog is a powerful language for trust management in networking. With extensions, we can support such basic functions that are needed for network policies as IP prefix range checks. However, it is not a universal solution for network policy management for all purposes. There are still limitations with Datalog that makes it less powerful or unsuitable for some applications.

- **Quantitative resource management.** We can use Datalog to certify resource allocations. But in nature, Datalog alone is not suitable for quantitative resource allocation problems that require numerical computations. We cannot compute solutions for resource allocations given the resource availability and demands with Datalog. Constraint logic programming as used in (for example) Anzere [74] is more powerful in such applications.
- **Reasoning at set level.** With Datalog, we can define nested sets/groups and reason if an element is a member of a group. However, Datalog doesn't support reasoning at set level. For example, we cannot reason if a set is a subset of another set. In resource or privilege delegation over sets of objects, we can determine if the ownership

of each object is valid with membership queries. But we cannot directly reason if the delegation of sets of objects is legal (i.e. the delegator has the privilege over all objects in the delegated set). We can use description logic for such applications.

- **Regular expressions.** Some policy management system uses regular expressions in the policies, including Merlin [78] and RPSL [6]. They use regular expressions (regex) to express such constraints as sequences of network functions or sequences of NSPs in the path. We can add built-in to support for naive matching with regex when it requires no logic reasoning to determine if each element in the queried list matches with the pattern in the regex. For example, we can determine if the path "$\langle AS0, AS1, AS2\rangle$" matches with the regex "$AS0.*AS2$", but not with "tag0.*tag2" which requires further logic inference to determine if the queried AS has certain attributes required in the regex. And we may use Datalog to specify policies that serve similar purposes as the regex. But in general, we cannot do regex matching that requires logic reasoning in Datalog.

## 7.2 Compensations and Motivations for NSPs

It needs motivations and incentives for the NSPs to enforce the customer routing policies as it requires effort and more hardware resources (e.g., more flow entries). The customer networks can pay to those NSPs for enforcement of their custom routing policies in a similar way as they pay for the transit service. We leave the business model for custom routing policies as future work.

Enabling customers to select NSPs for their traffic with customized policies will motivate the competition between NSPs. By decoupling infrastructure provider and network service provider, there will be various NSPs providing various network services for customers to choose from. This follows the "design for choice" principle and will ease the tussle in the Internet [28].

## 7.3 Intradomain Traffic Engineering for NSPs

NIaaS testbeds allow tenants to provision virtual service networks and manage the virtual service networks elastically when demands or available resources change. Different from existing approaches [52, 79, 87, 43, 49, 57, 44] that optimize the intradomain traffic engineering to meet the demands given existing infrastructures, virtual NSPs on testbeds allow operators to provision network resources and optimize the traffic engineering dynamically and jointly.

How to optimize the resource provision and intradomain traffic engineering to meet tenant demands with service level objectives about bandwidth, latency, loss, availability and network functions while minimizing the cost for network resources is challenging. We leave this problem for future work.

# 8

# Conclusion

In this dissertation, we aimed to enable programmable, elastic , flexible and secure multidomain networking over programmable network infrastructures, following the pluralist philosophy. We propose ExoPlex controller framework for deploying elastic virtual NSPs in network testbeds, with a focus on managing the security of multidomain networking with logical trust.

SAFE logical trust framework addresses the challenges in managing identity, resource access, naming, and network policies for multidomain networking. The Datalog logic language is declarative, rigorous, flexible and comprehensive to define network policies. We present off-the-shelf logic templates for some common services like group services and resource delegations. We also survey current approaches for Internet security like RPKI, BGPsec and MANRS with logic. We evaluate the performance of logical trust for different applications and showed that logical trust is fast enough and applicable. In practice, proper context pruning and indexing in the inference procedure can improve the inference performance greatly, which we should take into consideration when writing logic rules and designing the certificate linking structure.

The logical trust framework accommodates flexibility for customized and innovative net-

work policies. The logical peering model with exemplary path control policies demonstrates the potential for automatic and programmatic policy-based interdomain networking. We also address the deployment of such customized network policies in NSPs with ExoPlex. ExoPlex has the inference procedure to enforce such path control policies of both the source and destination that the traffic between their subnets passes through only paths that are compliant to the policies. ExoPlex can be the basis for a "testbed for trust" for interdomain networks that are constructed on the fly and span multiple slices, testbeds, and campuses. NSP owners and customers may experiment with policy for peering, routing, path control, and governance by specifying custom policies in logic, without changing the ExoPlex code. In particular, the trust plane supports customer policies for permissioning the NSPs themselves, so that customer traffic does not pass through untrusted NSPs (path control). A secure foundation with at least these features is a necessary prerequisite for safe testbed opt-in by real customer traffic—an important aspirational goal.

We survey prior work in route filtering and anti-spoofing and propose SCIF for more secure routing and ingress filtering. We compare the strength of different approaches within the same logical context, which gives us a better understanding of them and directions for improving them. SCIF allows customer networks and NSPs to jointly control the propagation of route advertisements with customized routing policies. By propagating routes only to the intended scope and trusted NSPs, we prevent ineligible NSPs that are not trusted by the source or the destination network from being able to spoof traffic for the prefix pair.

# Bibliography


[1] Esnet on-demand secure circuits and advance reservation system (oscars). `http://www.es.net/oscars/`.

[2] External layer2 connections (stitching). `https://chameleoncloud.readthedocs.io/en/latest/technical/networks/networks_stitching.html`.

[3] Mutually Agreed Norms for Routing Security (MANRS). `https://www.manrs.org/isps/guide/antispoofing/`.

[4] RIPE NCC RPKI Validator 2.21. `https://www.ripe.net/manage-ips-and-asns/resource-management/certification/tools-and-resources`.

[5] Martín Abadi and Boon Thau Loo. Towards a declarative language and system for secure networking. In *Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases (NETDB)*, pages 2:1–2:6. USENIX Association, 2007.

[6] Cengiz Alaettinoglu, Curtis Villamizar, Elise Gerich, David Kessens, David M Meyer, Tony Bates, Daniel Karrenberg, and Marten Terpstra. Routing policy specification language (RPSL). *RFC*, 2622:1–69, 1999.

[7] David G Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet Protocol (AIP). In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 339–350, 2008.

[8] Katerina Argyraki and David R Cheriton. Loose source routing as a mechanism for traffic policies. In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 57–64, 2004.

[9] Fred Baker and Pekka Savola. Ingress filtering for multihomed networks. Technical report, BCP 84, RFC 3704, March, 2004.

[10] Ilya Baldin, Jeff Chase, Yufeng Xin, Anirban Mandal, Paul Ruth, Claris Castillo, Victor Orlikowski, Chris Heermann, and Jonathan Mills. ExoGENI: A multi-domain infrastructure-as-a-service testbed. In *The GENI Book*, pages 279–315. Springer International Publishing, 2016.

[11] Ilya Baldin, Shu Huang, and Rajesh Gopidi. A Resource Delegation Framework for Software Defined Networks. In *Proceedings of the third Workshop on Hot Topics in*

*Software Defined Networking*, pages 49–54. ACM, 2014.

[12] I. Baldine, Y. Xin, A. Mandal, C. Heermann, J. Chase, V. Marupadi, A. Yumerefendi, and D. Irwin. Autonomic cloud network orchestration: A GENI perspective. In *2nd International Workshop on Management of Emerging Networks and Services (IEEE MENS '10), Co-Located with GLOBECOM'10*, Dec. 2010.

[13] Ilia Baldine, Yufeng Xin, Anirban Mandal, Paul Ruth, Aydan Yumerefendi, and Jeff Chase. ExoGENI: A multi-domain infrastructure-as-a-service testbed. In *International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, June 2012.

[14] Barnstormer Softworks. Welcome to geni-lib documentation! `https://geni-lib.readthedocs.io/en/latest/index.html`.

[15] Simon Bauer, Daniel Raumer, Paul Emmerich, and Georg Carle. Behind the scenes: what device benchmarks can tell us. In *Proceedings of the Applied Networking Research Workshop*, pages 58–65, 2018.

[16] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI veritas: Realistic and controlled network experimentation. In *SIGCOMM*, pages 3–14, 2006.

[17] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61, March 2014.

[18] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[19] Marshall Brinn, Nicholas Bastin, Andy C Bavier, Mark Berman, Jeffrey S Chase, and Robert Ricci. Trust as the foundation of resource exchange in GENI. *ICST Trans. Security Safety*, 2(5):e1, 2015.

[20] Milind M Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast layer-4 switching. In *International Workshop on Protocols for High Speed Networks*, pages 25–41. Springer, 1999.

[21] Randy Bush and Rob Austein. The Resource Public Key Infrastructure (RPKI) to router protocol. RFC 6810, RFC Editor, July 2013.

[22] Qiang Cao, Vamsi Thummala, Jeffrey S. Chase, Yuanjun Yao, and Bing Xie. Certificate linking and caching for logical trust. `http://arxiv.org/abs/1701.06562`, 2016. Duke University Technical Report.

[23] Qiang Cao, Yuanjun Yao, and Jeff Chase. A logical approach to cloud federation. *arXiv preprint arXiv:1708.03389*, 2017.

[24] Qiang Cao, Yuanjun Yao, and Jeffrey S. Chase. A logical approach to cloud federation. `http://arxiv.org/abs/1708.03389`, 2017. Duke University Technical Report.

[25] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4):1270–1283, 2009.

[26] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[27] Jeff Chase and Ilya Baldin. A retrospective on ORCA: Open Resource Control Architecture. In *The GENI Book*, pages 127–147. Springer International Publishing, 2016.

[28] David D Clark, John Wroclawski, Karen R Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's Internet. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 347–356, 2002.

[29] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe, and Andrew Warfield. Plutarch: An argument for network pluralism. *SIGCOMM Computer Communication Review*, 33:258–266, October 2003.

[30] John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113. IEEE, May 2002.

[31] Zhenhai Duan, Xin Yuan, and Jaideep Chandrashekar. Constructing inter-domain packet filters to control IP spoofing based on BGP updates. In *INFOCOM*, 2006.

[32] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693 (Experimental), September 1999.

[33] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[34] Nick Feamster, Lixin Gao, and Jennifer Rexford. How to lease the Internet in your spare time. *SIGCOMM Computer Communication Review*, 37:61–64, January 2007.

[35] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM computer communication review*, volume 43, pages 327–338. ACM, 2013.

[36] Paul Ferguson and Daniel Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. BCP 38, May 2000.

[37] Lixin Gao and Jennifer Rexford. Stable Internet routing without global coordination. *IEEE/ACM Transactions on networking*, 9(6):681–692, 2001.

[38] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2014.

[39] Phillipa Gill, Michael Schapira, and Sharon Goldberg. A survey of interdomain routing policies. *Computer Communication Review*, 44(1):28–34, 2014.

[40] Sharon Goldberg, Michael Schapira, Peter Hummon, and Jennifer Rexford. How secure are secure interdomain routing protocols. *ACM SIGCOMM Computer Communication Review*, 40(4):87–98, 2010.

[41] Arpit Gupta, Nick Feamster, and Laurent Vanbever. Authorizing network control at software defined Internet exchange points. In *Proceedings of the Symposium on SDN Research*, page 16. ACM, 2016.

[42] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined Internet exchange. In *ACM Conference on SIGCOMM*, pages 551–562, 2014.

[43] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[44] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87. ACM, 2018.

[45] Jon Howell and David Kotz. End-to-end authorization. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 11–11, Berkeley, CA, USA, 2000. USENIX Association.

[46] Geoff Huston. BGP in 2018. `https://blog.apnic.net/2019/01/16/bgp-in-2018-the-bgp-table/`.

[47] Iperf - the network bandwidth measurement tool. `https://iperf.fr/`.

[48] Jacks. `https://www.emulab.net/protogeni/jacks-doc/`.

[49] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[50] Fan Jiang, Claris Castillo, and Charles Schmitt. RADII: Bridging the divide between

data and infrastructure management to support data driven collaborations. In *Proceedings of IEEE BIgData Conference*, 2016.

[51] Trevor Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115. IEEE, May 2001.

[52] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 253–264. ACM, 2005.

[53] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure Border Gateway Protocol (S-BGP)–Real world performance and deployment issues. In *Proc. of Network and Distributed System Security Symposium,(San Diego, California)*, 2000.

[54] Stephen Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (s-bgp). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[55] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 271–282. ACM, 2014.

[56] R Kisteleki and B Haberman. Securing routing policy specification language (RPSL) objects with resource public key infrastructure (RPKI) signatures. Technical report, Technical report, RFC Editor. RFC7909.(DOI: 10.17487/RFC7909). Acessado em 03/02, 2019.

[57] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 1–14. ACM, 2015.

[58] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and adoptable source authentication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, volume 8, pages 365–378, 2008.

[59] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11), 2009.

[60] Ratul Mahajan, David Wetherall, and Thomas E Anderson. Mutually controlled routing with independent ISPs. In *NSDI*, 2007.

[61] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pages 51–58. IEEE, 1998.

[62] J. Mambretti, J. Chen, and F. Yeh. Next generation clouds, the Chameleon Cloud testbed, and Software Defined Networking (SDN). In *International Conference on*

*Cloud Computing Research and Innovation (ICCCRI)*, pages 73–79, Oct 2015.

[63] Rick McGeer, Mark Berman, Chip Elliott, and Robert Ricci, editors. *The GENI Book*. 2016.

[64] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, Recommendations of the National Institute of Standards and Technology, September 2011.

[65] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with ICING. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 30. ACM, 2011.

[66] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM Computer Communication Review*, 33(1):59–64, 2003.

[67] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[68] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[69] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[70] Sophie Y Qiu, Patrick D McDaniel, and Fabian Monrose. Toward valley-free interdomain routing. In *2007 IEEE International Conference on Communications*, pages 2009–2016. IEEE, 2007.

[71] QRATOR Labs blogpost. `https://radar.qrator.net/blog`.

[72] RADb. `https://www.radb.net/`.

[73] Riak key value store. `http://docs.basho.com/riak/`, 2016.

[74] Oriana Riva, Qin Yin, Dejan Juric, Ercan Ucan, and Timothy Roscoe. Policy expressivity in the Anzere personal cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 14. ACM, 2011.

[75] Timothy Roscoe. The end of Internet architecture. In *Proceedings of the 5th Workshop on Hot Topics in Networks*. Irvine, CA, USA, 2006.

[76] Paul Ruth and Mert Cevik. Experimenting with AWS Direct Connect using Chameleon, ExoGENI, and Internet2 Cloud Connect. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

[77] Alan Shieh, Emin Gün Sirer, and Fred B Schneider. NetQuery: A knowledge plane for reasoning about network properties. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 278–289. ACM, 2011.

[78] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with Merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 24. ACM, 2013.

[79] Ashwin Sridharan, Roch Guérin, and Christophe Diot. Achieving near-optimal traffic engineering solutions for current OSPF/IS-IS networks. *IEEE/ACM Transactions On Networking*, 13(2):234–247, 2005.

[80] Kotikalapudi Sriram and Matthew Lepinski. BGPsec protocol specification. RFC 8206, September 2017.

[81] Kotikalapudi Sriram, Doug Montgomery, D McPherson, Eric Osterweil, and Brian Dickson. Problem definition and classification of BGP route leaks. *RFC 7908*, 2016.

[82] Styla. `https://github.com/fedesilva/styla`.

[83] Jeroen van der Ham, Paola Grosso, Ronald van der Pol, Andree Toonk, and Cees de Laat. Using the Network Description Language in optical networks. In *Tenth IFIP/IEEE Symposium on Integrated Network Management*, May 2007.

[84] Anduo Wang, Zhijia Chen, Tony Yang, and Minlan Yu. Enabling policy innovation in interdomain routing: A software-defined approach. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 62–68, 2019.

[85] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, 2018.

[86] Yufeng Xin, Ilya Baldin, Anirban Mandal, Paul Ruth, and Jeff Chase. Towards an experimental legoland: Slice modification and recovery in ExoGENI testbed. In *Testbeds and Research Infrastructures for the Development of Networks and Communities*, pages 35–45. 2016.

[87] Dahai Xu, Mung Chiang, and Jennifer Rexford. Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering. *IEEE/ACM Transactions on networking*, 19(6):1717–1730, 2011.

[88] Wen Xu and Jennifer Rexford. MIRO: multi-path interdomain routing. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 171–182, 2006.

[89] Xiaowei Yang, David Clark, and Arthur W Berger. NIRA: a new inter-domain routing architecture. *IEEE/ACM Transactions On Networking*, 15(4):775–788, 2007.

[90] Yuanjun Yao, Qiang Cao, Jeff Chase, Paul Ruth, Ilya Baldin, Yufeng Xin, and Anirban Mandal. Slice-based network transit service: Inter-domain L2 networking on ExoGENI. In *INFOCOM Workshops (DCC)*, pages 736–741. IEEE, 2017.

[91] Yuanjun Yao, Qiang Cao, Rubens Farias, Jeff Chase, Victor Orlikowski, Paul Ruth, Mert Cevik, Cong Wang, and Nick Buraglio. Toward live inter-domain network services on the ExoGENI testbed. In *INFOCOM Workshops (CNERT)*, pages 772–777. IEEE, 2018.

[92] Yuanjun Yao, Qiang Cao, Paul Ruth, Mert Cevik, Cong Wang, and Jeff Chase. Logical peering for interdomain networking on testbeds. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 824–829. IEEE, 2020.

[93] Dapeng Zhu, Mark Gritter, and David R Cheriton. Feedback based routing. *ACM SIGCOMM Computer Communication Review*, 33(1):71–76, 2003.

# Biography

Yuanjun Yao was born in an ethnic minority region in Enshi, Hubei, China. His research interests are network security, computer networking and distributed systems. He defended his PhD thesis at Duke University in October 2020. Before that, he received a Bachelor of Engineering in Automation from Tsinghua University in 2015.